A Parallel Cellular Automaton Simulation Framework using CUDA

by

Ryno Fourie

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science in the Faculty of Science at Stellenbosch University

Department of Mathematical Sciences,
Computer Science Division,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Prof. L. van Zijl

March 2015

Stellenbosch University https://scholar.sun.ac.za

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: 2015/02/18

Copyright © 2015 Stellenbosch University
All rights reserved.

Stellenbosch University https://scholar.sun.ac.za

Abstract

A Parallel Cellular Automaton Simulation Framework using CUDA

R. Fourie
Department of Mathematical Sciences,
Computer Science Division,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MSc (Computer Science)
February 2015

In the current digital age, the use of cellular automata to simulate natural systems has grown more popular as our understanding of cellular systems increases. Up until about a decade ago, digital models based on the concept of cellular automata have primarily been simulated with sequential rule application algorithms, which do not exploit the inherent parallel nature of cellular automata. However, since parallel computation platforms have become more commercially available, researchers have started to investigate the advantages of parallel rule application algorithms for basic cellular automata.

For this thesis, a parallel cellular automaton framework, based on NVIDIA CUDA is developed to simplify the implementation of a wide range of cellular automata. This framework is used to investigate the potential performance advantages of using graphical processing units as a parallel processing platform for cellular automata.

Stellenbosch University https://scholar.sun.ac.za

Uittreksel

’n Parallelle Sellulêre Outomaat Simulasieraamwerk gebaseer op CUDA

R. Fourie

Departement Wiskundige Wetenskappe,
Afdeling Rekenaarwetenskap,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MSc (Rekenaar Wetenskap)

Februarie 2015

In die huidige digitale era het die gebruik van sellulêre outomate om natuurlike stelsels te simuleer, aansienlik toegeneem soos wat ons begrip van sellulêre stelsels verbreed word. Tot om en by ’n dekade gelede is digitale modelle wat met behulp van sellulêre outomate gesimuleer word, hoofsaaklik met sekwensiële reëlfunksies gesimuleer. As gevolg hiervan het die inherente parallelle natuur van sellulêre outomate nie tot sy volle reg gekom nie. Aangesien parallelle berekenings-platforms egter onlangs meer kommersieël beskikbaar geraak het, span navorsers hierdie platforms nou in om parallelle reëlfunksies te skep vir meer basiese sellulêre outomate.

Vir hierdie tesis is ’n parallelle sellulêre outomaat simulasieraamwerk geskep, wat gebruik maak van die NVIDIA CUDA parallelle berekenings-platform. Hierdie raamwerk is geskep om die implementasie van ’n verskeidenheid van sellulêre outomate te vereenvoudig, en is ingespan om die potensiële tydsvoordeel van grafiese verwerkingseenhede te ondersoek in die implementasie van sellulêre outomate.

Stellenbosch University https://scholar.sun.ac.za

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations:

- my supervisor, professor Lynette van Zijl to whom I am grateful for her invaluable input;
- Naspers and the MIH Media Lab, that provided me with financial support while I conducted my research and wrote my thesis;
- John Owens and David Luebke, the instructors of the Udacity course CS344: Intro to Parallel Programming;
- NVIDIA;
- and finally, to my family and to God, for their support and guidance during the last two years.

Stellenbosch University https://scholar.sun.ac.za

Contents

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

List of Figures

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

List of Tables

Stellenbosch University https://scholar.sun.ac.za

Chapter 1

Introduction

Computer simulations play a large part in modern life. Physical and biological research is supported by various mathematical and computer science techniques and programming algorithms, to get a more concise picture of how these systems work or would react in certain scenarios [61].

To gain an understanding of biological self-replicating systems, the concept of cellular automata (CA) was introduced by Ulam and Von Neumann in the 1940s [58]. A CA is made up of an N-dimensional grid of discrete cells, where the state of each cell changes as a function of time, according to a predefined set of rules [24]. Each cell contains a value that represents the current state of a cell, where, on the most basic level, the state will either be 0 or 1 (dead or alive) [45, 59].

There are two key aspects to take into account regarding CA. Firstly, a cell generally does not have a time dependency on any of its neighbouring cells during the calculation of its next state. And secondly, the next state of each cell is calculated before the entire CA is updated. Thus, the cells of a CA are essentially processed in a parallel fashion. In practice, however, the general trend has long been to use sequential algorithms to simulate a CA, where only one cell is processed at a time. For a two-dimensional CA this practice is equal to a $O(MN)$ time complexity, where M is the number of rows and N is the number of columns of the CA grid. A compute-intensive rule application function will exacerbate the high execution time.

A proper parallel updating algorithm for the CA would potentially alleviate the execution time problem. In order to simulate a CA with a parallel algorithm, a parallel processing platform is required. Ideally, it should maximize the number of cells that are processed simultaneously. With the advancements in graphical processing unit (GPU) architecture in the past decade, that allow programmers to harness the power of the GPU for general computation, the GPU seems like an ideal platform on which a CA can be simulated in parallel.

In this thesis the performance gain achieved by implementing a CA on a GPU is investigated, by using different parallel data segmentation methods. Each data

Stellenbosch University https://scholar.sun.ac.za

segmentation method uses a different amount of GPU processing resources. To that end, we develop a general CA framework in C/C++, based on CUDA. The framework will be used to investigate the execution times of the different parallel CA simulations based on the data segmentation methods, in comparison to the results obtained with a sequential CA simulation. All parallel algorithms will be performed on the GPU, and all the sequential algorithms will be performed on the CPU. To set a benchmark for a basic CA simulation, the Game of Life CA will be implemented. For more complex CA, the concept of virtual clay deformation will be implemented as a benchmark, since more complex calculations need to be performed when calculating a new generation. And to set a benchmark for CA simulations that have potential race conditions for a parallel implementation, the ant clustering CA will be implemented.

1 Thesis outline

In this thesis, different parallel processing platforms that have been applied for CA simulations, are investigated and discussed in Chapter 2.

In Chapter 3 the requirements, design, and implementation of the CA framework is discussed.

The application of the CA framework is discussed in Chapter 4, where the performance of sequential and parallel CA rule application algorithms are investigated.

In Chapter 5 we conclude this thesis by discussing the overall findings and by stating possible future work.

Stellenbosch University https://scholar.sun.ac.za

Chapter 2

Literature survey

The framework developed for this thesis is applied to investigate the potential performance advantages of parallel implementations of CA. As background, different applications of CA are introduced. This is followed by an overview of parallelization techniques for solving CA, with specific reference to the graphics processor. Finally, an overview of GPGPU is given, how this field has expanded, and where it has been applied.

1 Cellular automata

The concept of discrete CA models was introduced in the 1940s, and interest in it has grown since then [22, 43, 45, 60]. The use of CA to solve small scale practical problems has increased in a number of fields. For example, Nandi et al apply CA in the field of cryptography to construct one-dimensional block-ciphers [34]; Yuen et al discuss traffic modeling and structural design using CA [65]; Tran et al discuss a CA model to simulate the cardiac tissue electrical activity observed in patients that suffer from atrial fibrillation [53]; and Gosálvez et al apply CA for erosion simulation using both continuous and discrete CA models [18]. Other applications of CA are listed by Ganguly et al [13], and include:

- simple simulations such as Conway's Game of Life, the firing squad problem, and the iterated prisoner's dilemma;
- social systems, such as social interactions and racial segregation;
- modeling physical and biological systems such as water-flow modeling, chemical processes and the immune system;
- Very Large Scale Integration (VLSI);
- digital image processing;

Stellenbosch University https://scholar.sun.ac.za

- pattern recognition; and
- the theoretical inspiration of CA for the development of parallel processors.

An important attribute of CA is its inherent parallel nature; cells can generally be processed in parallel without the risk of causing an error in the overall state of the CA. Tran et al reason that simulating cardiac tissue electrical activity using a CA model would grant them better performance, when coupling the CA model with a parallel adapted algorithm. This is in contrast to the conventional method of cardiac tissue simulation, which involves solving elaborate systems of partial differential equations, which would normally be sent off to supercomputer facilities [53].

To look beyond the CPU and its sequential nature of processing, research has focused on developing parallelization techniques based on parallel processors. The next section will look at parallelization techniques and the related hardware that has been applied to CA algorithms.

2 Parallelization techniques

In this section different parallel processing platforms that have been used to implement CA are investigated.

2.1 Field Programmable Gate Array

Murtaza et al proposed to address the problem of sequential processing of CA by using the Field Programmable Gate Array (FPGA) [33]. Their study was conducted in order to provide an alternative method to process CA in a more parallel fashion than can be achieved with a single CPU.

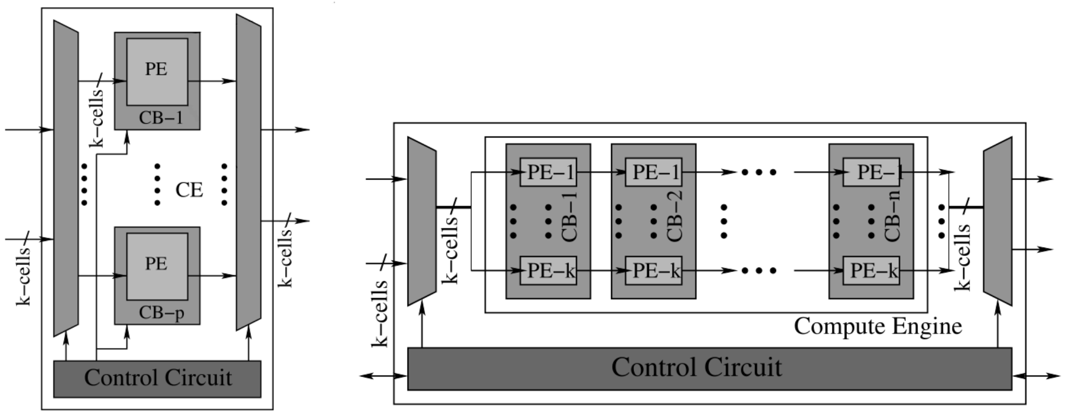
An FPGA consists of a number of compute engines, RAM and I/O ports, where each compute engine has a set number of processing elements. Two drawbacks of FPGAs are that these devices are generally expensive, and, because the architecture of the FPGA is based on programmable logic blocks and not a fixed instruction-set (as is the case with CPUs, GPUs and MPPAs[1]), it requires knowledge of logic programming to be able to use FPGAs effectively [48]. However, current FPGAs are powerful computational devices, and can perform over 10^{12} single precision floating point operations per second [1].

At an abstract level, an FPGA is the same as a CA. During execution of a CA algorithm, each cell is mapped to a processing element in the FPGA. Each processing element then calculates the next state of the cell assigned to it. If the CA has more cells than there are processing elements on the FPGA, any unprocessed cells will be assigned to the next available processing element.

[1]Massively Parallel Processor Array.

Stellenbosch University https://scholar.sun.ac.za

Murtaza et al considered both compute intensive CA and I/O intensive CA. For each case, a different FPGA board is proposed, to alleviate potential bottlenecks as best possible. For a compute intensive CA, more time is spent calculating the next state of a cell, and thus an FPGA board that can utilize more of the available compute resources, is used. The abstract model shown in Figure 2.1(a) depicts this scenario: the data of k cells can be processed at a time, and thus k processing elements are assigned to process the data and write the results. For an I/O intensive CA, more time is spent reading data into memory. To compensate for this bottleneck, an FPGA board with a more sophisticated control block (which is responsible for I/O, and routing data to processing elements) is used. This allows the FPGA to chain the results of n processing elements together in order to calculate the n'th generation of a single cell. This process is applied to all cells currently in memory, and after completion, the processed cells are written to storage, and the next chunk of unprocessed cells is loaded into memory (see Figure 2.1(b)).



(a) Compute-bound board (b) I/O-bound board

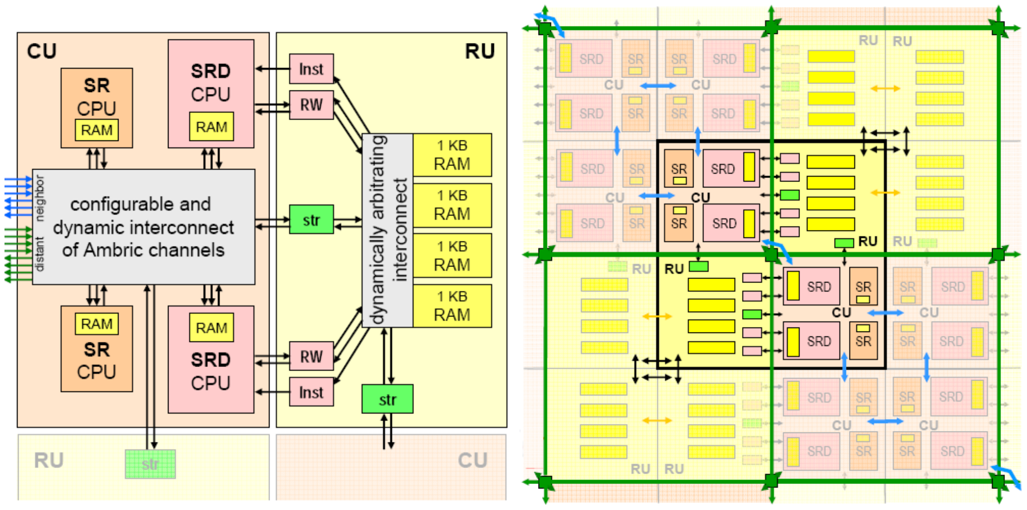
Figure 2.1: An abstract representation of compute-bound and I/O-bound FPGA boards. The compute-bound board scales vertically: it assigns more resources to compute one generation of k cells in parallel. The I/O-bound board scales horizontally: it assigns more resources to process n generations of k cells in parallel (taken from [33]).

2.2 Massively Parallel Processor Array

The second parallelization technique uses the more recent developed Massively Parallel Processor Array (MPPA). The MIMD (Multiple Instruction, Multiple Data) architecture of the MPPA is a single chip, comprising hundreds of 32-bit RISC processors, assembled onto compute units and laid out in a grid. These compute units are interconnected, allowing the MPPA to distribute a workload among the

Stellenbosch University https://scholar.sun.ac.za

processors, which then process the data independently, in a parallel fashion. This design makes the MPPA a suitable alternative for processing CA. For example, the Am2045 MPPA, one of the first commercially launched MPPAs, developed by Ambric Inc., consists of 42 CU–RU (Compute Unit and RAM Unit) brics[2], where each bric has two CUs and two RUs. The Am2045 MPPA chip has a total of 336 RISC processors [6]. Figure 2.2 gives a schematic overview of the Ambric Am2045 MPPA.



(a) CU and RU bric (b) Bric-interconnections

Figure 2.2: Schematic representations of the Ambric Am2045 CU and RU bric, which show how the brics are interconnected on the chip (taken from [19]).

Since an MPPA consists of a multitude of RISC processors, these processors are not clocked at a high frequency, in order to reduce the power usage of an MPPA. Although MPPAs have efficient performance based on power consumption, it does lack in terms of cache size per processor, and in terms of total global memory. These limitations, as well as the manner in which the compute units are interconnected, can introduce problems when designing algorithms for MPPAs [48].

Millo et al discuss the similarities between CA and MPPAs, and how to design a schema to optimally map a CA to an MPPA [32]. The biggest issue identified and addressed is the routing of data between the multitude of parallel-execution cores on an MPPA. Millo et al first looked at implementing a k-periodically routed graph (KRG), that allows data-flow routing directives to occur, and introduced the Neighbourhood Broadcasting Algorithm (NBA) which calculates these routing

[2]A bric is the term used by Ambric Inc. to refer to the dies onto which all the physical processing units are assembled. These brics are laid out in a grid, to make up a complete MPPA unit.

Stellenbosch University https://scholar.sun.ac.za

directives. By combining the NBA with a KRG for a specific CA, the data of a cell is correctly propagated to its neighbourhood, thereby increasing the efficiency of calculating the next generation of a CA.

A study conducted by Top et al tests the programmability and performance of both an FPGA and an MPPA [51]. Their study was based on an Altera StratixII FPGA[3] and an Ambric Am2045 MPPA. Top et al conclude that mapping algorithms onto the FPGA is more difficult than for the MPPA, as it requires significant knowledge of the hardware, software, and parallel computation techniques. The algorithms implemented for the FPGA take longer to compile and are more difficult to debug than when compared to the same algorithms implemented for the MPPA. However, the FPGA outperforms the MPPA by a factor of 3 to 11 times in execution speed. The MPPA did however outperform the sequential implementations (executed on the benchmark system which uses a dual-core CPU, clocked at 3.4GHz), and produced a speed up of 2 to 9 times the execution speed of the sequential algorithm.

2.3 GPU

The third parallelization technique is based on the GPU. Navarro et al focused on the performance gains offered by the GPU and the CPU [35]. Their study shows that the modern CPU, based on the MIMD architecture, is more adept at running different processes, each with its own data set, in parallel. Computing scalable problems, normally consisting of a single process with a large data set, is where the GPU excels, as the SIMD (Single Instruction, Multiple Data) architecture of the GPU is more suitable for problems of this kind. The GPU architecture is well suited to process systems based on CA, since CA are essentially also based on SIMD; a predefined set of rules is applied on a fixed number of cells for each time step.

Kauffmann et al conducted two related studies to investigate the computational power of the GPU as applied to CA based problems. In both of these studies, the GPU is used to process image segmentation algorithms, with medical image data represented as CA. In the first study, the watershed-transformation is used to identify image segments [23]. The Ford-Bellman shortest path algorithm (FBA) is then applied to perform the segmentation. For their second study, a GPU based framework is implemented to perform a seeded N-dimensional image segmentation, again using the FBA [24]. The results obtained in both studies show that the GPU implementation outperforms a similar sequential algorithm executed on a CPU, as the image size increases. The parallel implementation on the GPU is about ten times faster than the sequential implementation.

Kauffmann et al noted two problems with the GPU implementation. The first issue had to do with slow memory transfer from the CPU to the GPU, across

[3]Altera Corporation is one of the current FPGA market leaders.

Stellenbosch University https://scholar.sun.ac.za

the PCI-Express port, which hampered overall performance. The other issue was due to restricted data-type usage on the GPU, where double-precision floats are unavailable on the GPU that was used for their experiments.

Gobron et al conducted a study on artificial retina simulation, and proposed a pipelined model of an artificial retina based on CA. To simulate the CA, a parallel algorithm for the GPU was proposed. Their study shows that the parallel implementation is about 20 times faster than the conventional sequential implementation [17]. In a subsequent study by Gobron et al, a GPU accelerated method for real-time simulation of a two-dimensional hexagonal-grid CA, was proposed. In this study, six different computer configurations were used, and in all cases, the parallel implementation outperforms the sequential implementation [16]. For both studies, the OpenGL Shading Language (GLSL) was used to implement the parallel algorithms.

Zaloudek et al looked at evolutionary CA rules, specifically focusing on genetic algorithms, and used the NVIDIA CUDA framework (refer to Section 4.2, page 11) to implement parallel algorithms, that execute these rules on one-dimensional CA [66].

Rybacki et al looked at basic two-dimensional CA used in social sciences, including Game of Life, the parity model, and the majority model. The James II Java simulation and modeling framework was used to measure the throughput of several CPU compute models and a GPU compute model, when used to simulate the CA [42].

Ferrando et al expanded on a study conducted by Gosálvez et al [18], and used an octree data structure to store surface model data. Each octree is stored in a "supercell" and a surface is modeled with continuous CA that is made up of a two-dimensional grid of supercells. The algorithm proposed to simulate complex surface erosion, loads data into the GPU global memory and uses a CUDA kernel function to process the data [12].

Caux et al used CUDA to accelerate the simulation of three-dimensional CA models based on integrative biology. Two parallel implementations, which compare global and local memory usage, were measured against a sequential implementation. Their study shows that both parallel implementations produce a significant speed up over the sequential implementation [7].

López-Torres et al looked at a CA model used to simulate laser dynamics, and presented a parallel implementation using CUDA. The parallel implementation delivers a speed up of up to 14.5 over the sequential implementation [29].

Finally, a study conducted by Gibson et al investigates the speed up of threaded CPU and GPU implementations of the Game of Life CA, over a sequential implementation. In their study, different configurations of a two-dimensional Game of Life CA were used. The study also looked at the difference in performance when using different work-group (thread-block) sizes. OpenCL was used for the GPU implementations [15].

Stellenbosch University https://scholar.sun.ac.za

Of the three platforms discussed above, the GPU is the most viable option in general, since it is the most accessible of the three and still provides excellent performance. Its design is well suited for problems modeled with CA, and a GPU is much cheaper than an FPGA or MPPA.

The next two sections give an overview of how the GPU has been adapted to solve more general computational problems, as well as the programming platforms used to harness the power of the GPU.

3 GPGPU overview

Over the last 15 years, GPU manufactures started shifting the focus of graphics processor architectures away from a fixed-function pipeline and started placing more emphasis on versatility (refer to Appendix A on page 95). This shift in the architectural design allowed more focus on general-purpose programming on the GPU (GPGPU). The earliest GPGPU studies, all performed on consumer GPUs, focused on a variety of areas, including: basic mathematical calculations [54], fast matrix multiplications [28], image segmentation and smoothing [64], physical system modeling and simulations [20] and fluid dynamics [11]. Thompson et al discuss the development of a programming framework that allows for easier compilation of general algorithms to machine instructions executed by the GPU [49]. The framework was used to implement a matrix multiplication algorithm and a 3-satisfiability problem solver. Both implementations performed on the GPU delivered a substantial speed up in performance when compared to the sequential implementations.

Currently, GPGPU is regularly applied in medical image processing. Shi et al studied techniques used for medical image processing that are portable to the GPU (where parallelization is exploitable), and evaluated the performance of the GPU for these techniques [44]. For the three techniques (segmentation, registration, and visualization) studied, the GPU tended to show better performance than the CPU. Kirtzic et al used the GPU to reduce the latency on a system which simulates radiation therapy [26]. The experiments showed a substantial increase in performance when compared to both a sequential and threaded CPU implementation.

4 GPGPU APIs

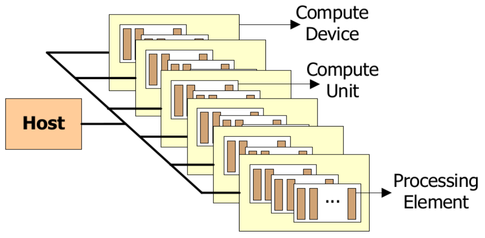
Owens et al [40] surveyed the evolution of GPGPU, and in particular the progression of the GPU's traditional fixed-function pipeline into a more flexible programmable pipeline. In order to exploit the changes made to GPU architectures, parallel programming APIs were developed. Two of the current most popular APIs are OpenCL [25] and CUDA [39].

Stellenbosch University https://scholar.sun.ac.za

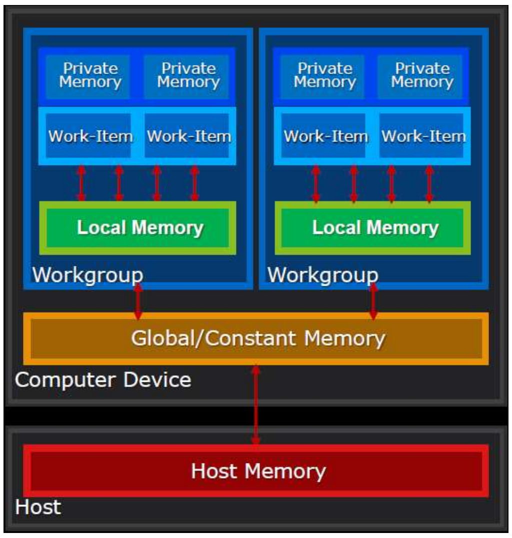
4.1 OpenCL

OpenCL (Open Computing Language) is an open-source low-level API used for heterogeneous computing. OpenCL 1.0, the first version of OpenCL, was introduced in 2008. The most recent version, OpenCL 2.0, was released in 2013. OpenCL is maintained by Khronos Group, and is used in gaming, entertainment, scientific research, and medical software. It is also the most widely used open-source parallel programming API [25].

OpenCL supports a diverse range of computational devices including CPUs, GPUs, FPGAs, and digital signal processors (DSPs). The core concept of OpenCL is to unify heterogeneous computing by employing all available compute resources and effectively balancing the system load. An abstract model of the compute platform sees a host device connected to compute devices. The host device is responsible for starting kernel processes that are executed by the compute devices. Compute devices are made up of compute units, each with its own set of processing elements (see Figure 2.3(a)). The host device also regulates data transfer from host memory to the so called global memory of a compute device. Global memory on the compute device is then transferred to the local or workgroup memory of each compute unit, which is used to store local variables and locally calculated results (see Figure 2.3(b)). In order to execute an OpenCL program, the host device has



(a) Compute platform model



(b) Memory transfer hierarchy

Figure 2.3: Abstract representations of the compute platform model and memory transfer hierarchy (taken from [55]).

to establish communication with OpenCL viable devices and create a context with which it will address these devices. Then, according to the problem being solved,

Stellenbosch University https://scholar.sun.ac.za

kernels are selected for aspects of the problem that can be solved in parallel. Data is transferred to the compute device(s) that will be employed to process the data (solve the problem), and the kernels are then started. After the work has been completed, the processed data is transferred back from the compute device(s) to the host device.

4.2 CUDA

CUDA (Compute Unified Device Architecture) is a proprietary parallel programing platform, developed by NVIDIA and is used with CUDA-enabled NVIDIA GPUs, that have been designed on the unified shader GPU architecture. CUDA was first introduced in 2006 as CUDA 1.0, when NVIDIA launched the GeForce G8x GPU series. The most recent production version of CUDA, CUDA 6.5, was released in 2014 [39]. CUDA is accessible through supported industry standard languages, including C, C++ and Fortran. Essentially, CUDA is a set of language-specific libraries, compiler directives, and extensions, compiled into an SDK. There are also third party wrappers available for other common programming languages including Python, Perl, Java, Haskell and MATLAB (amongst others).

On an abstract level, CUDA is similar to OpenCL. A kernel process is started by the host (CPU), after the relevant data has been transferred from host memory to device (GPU) memory, and is then executed by the device. Figure 2.4 gives an overview of this process. As with OpenCL, the data is transferred back to the host for post processing and analysis, after the kernel process has been completed. Work performed by the device is segmented into blocks of threads, or thread-

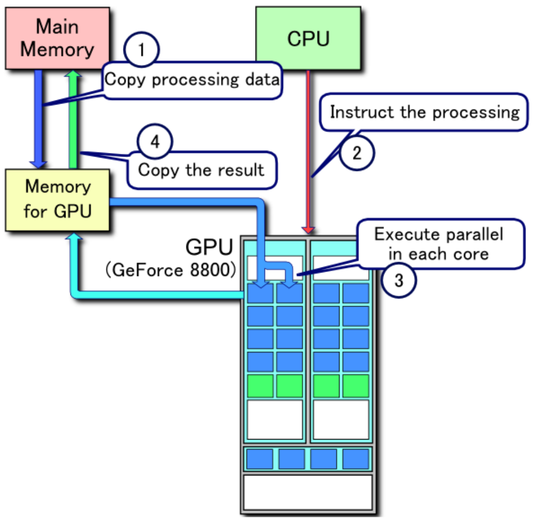


Figure 2.4: An abstract representation of the model of work flow, from the CPU (host) to the GPU (device) (taken from [52]).

Stellenbosch University https://scholar.sun.ac.za

blocks, and a grid of thread-blocks. Each thread-block is defined as a one, two, or three-dimensional array of threads, with the total number of threads not allowed to exceed 1024. Each thread-block is executed by one of the streaming multiprocessors (SMs), with a single GPU currently having from 2 to 15 SMs [5]. To optimize data processing on the device, the global memory of the device can be cached to each SM. This allows an SM to read and write relevant data more quickly during kernel execution.

CUDA is exclusively used for data parallelism with NVIDIA GPUs, whereas OpenCL is used for both data parallelism (with GPUs, DSPs, and FPGAs) and task parallelism (with multi core and multi-threaded CPUs). The limited device support of CUDA allows NVIDIA to include more intricate code debugging and runtime debugging tools, that work in tandem to point out errors in code (aside from syntax errors), such as incorrect host-to-device memory transfers [21, 27]. These tools along with other tools and features that have been added to CUDA since its original release in 2006 [38], help to simplify the task of the programmer when implementing parallel algorithms. A programmer also does not need to learn a new programming language if the programmer already has experience with an officially supported language. CUDA has also been used in more studies that investigate potential speed ups for CA simulations, and all these studies indicate a success when CUDA is used. All these advantages considered make CUDA an ideal platform to use for the CA framework implemented for this thesis.

5 Conclusion

In Section 2.3 a variety of GPU based CA studies are discussed. The studies that have been performed tend to investigate the speed up of a proposed parallel implementation of a *specific* CA, over a conventional sequential implementation.

In this thesis we investigate the potential performance advantages of parallel CA implementations, based on different parallel data segmentation methods; each uses a different amount of GPU processing resources (see Section 3.1.2 on page 22). The computation time of the different parallel CA implementations are measured and compared with the results of a sequential CA implementation. The investigation is performed on the Game of Life, clay deformation, and ant clustering CA; all implemented with the CUDA based CA framework developed for this thesis.

In the next chapter the design and implementation of the CA framework is discussed.

Stellenbosch University https://scholar.sun.ac.za

Chapter 3

Design and implementation

The CA framework proposed for this thesis encompasses many diverse concepts. Essentially the framework must allow a programmer to easily create a new CA, by simplifying the process of specifying unique attributes of the CA, specifying the rule system of the CA, and how the CA must display its overall state during a simulation. The main application of the framework for this thesis, will be to investigate the performance difference between sequential and parallel algorithms used to calculate the generation of a CA.

This chapter focuses on design and implementation aspects of the proposed CA framework. Essential requirements for the framework are discussed, followed by an overview of the structure of the framework, and how all the different parts must interact in order to simplify the end-user's experience when using the framework.

1 Requirements

Core attributes of the proposed framework include:

- modularity;
- abstraction;
- sequential and parallel algorithms;
- visualization; and
- experimental results.

In the following subsections these attributes will be discussed, as well as the requirements from a software development perspective, in order to incorporate these attributes into the framework.

Stellenbosch University https://scholar.sun.ac.za

1.1 Modularity

Modularity refers to software partitioning, and entails dividing different aspects of the software into separate modules [4]. When looking at the framework proposed in this thesis, different modules will be implemented based on the characteristics of CA.

Since a CA is a grid of cells, which changes according to a set of rules unique to the specific CA, the framework has to simplify the creation of a new and unique CA. CA are used to model a variety of problems, and therefore the different attributes of a CA such as its states, rules, and the dimensions of its grid, must be made adaptable based on the problem being modeled. A basic CA such as Game of Life only needs cells to keep record of their current states [45]. However, a CA such as is proposed for clay deformation, requires each cell to also keep record of how much clay the cell contains [3]. Each CA also requires a separate rule set and algorithm which applies the rule set. Based on these ideas, it will be ideal to design a framework for a CA based on three core modules: a module that defines the attributes of the cell, a module that combines cells into a grid, and a module that defines the rule set of the CA and how it is applied to the grid of cells.

In addition to these core modules, a module for rendering CA and a module for executing CA algorithms will also be required. The rendering module must be able to visualize the current state of CA, which generally means to draw a CA according to the specification of the programmer. The execution module should start the rule application algorithm of the CA. The execution module must also be able to interact with the rendering module, to allow a user to render the CA either perpetually, after a certain number of time steps, or not at all, depending on the experimental data being gathered during execution.

1.2 Abstraction

A modular approach to create the framework will help to reduce the coding process when implementing a new CA. By also making features of the proposed core modules abstract in nature (where it applies), the framework as a whole will be able to function more cohesively.

By creating a base abstract class that incorporates the base attributes and functions needed for a CA, a derived class for each unique CA can then be implemented. An example of an abstraction approach, along with the programming concept of polymorphism, occurs when the rule application function for a CA is called from the execution module. The execution module only needs one *central* function with an argument for the CA base class, which will allow it to call the rule application function of any derived CA class.

However, if an approach of abstraction is not followed, a separate function for each unique CA will need to be added to the execution module in order to call the

Stellenbosch University https://scholar.sun.ac.za

rule application function of each unique CA.

Abstraction is thus advantageous for modules that are extensible, and helps to reduce redundant code.

1.3 Sequential and parallel algorithms

The algorithm that applies the rules of a CA is generally unique. However, specific aspects of an algorithm for a certain CA A might coincide with the algorithm used for CA B, such as determining the neighbourhood of a cell. For any coinciding parts of algorithms, it would be advantageous to write static functions stored in a central module that can be called in order to provide the service. In doing so, re-occurring parts of different algorithms will not need to be re-coded for each separate algorithm.

For parallel algorithms, the CUDA parallel platform developed by NVIDIA will be used to apply the rules of a CA. CUDA follows a general protocol of transferring the required data to the GPU, and after having performed the relevant work on the data, the data is transferred back to main memory. This process of transferring data will have to be performed for any parallel algorithm, and must be made available to all CA as a general service.

All algorithms used for applying the rules of a CA must provide the time taken to complete the algorithm. This data is essential for analyzing performance differences between sequential and parallel algorithms.

1.4 Visualization

In order to simulate every new generation of a CA, visualization is an important requirement. Therefore a sufficient visual drawing library needs to be incorporated into the framework. Ideally this library must be able to render the state of each cell in the CA. It would be preferable if the library provides features to create a Graphical User Interface (GUI), to simplify interaction with the CA, and to display experimental results.

Notice that the process of visualizing each new generation does influence the total time taken to complete an experiment, since an additional constant amount of time is required to render each generation of the CA. For parallel algorithms, the data processed on the GPU must be transferred back to main memory in order to render the next generation of the CA. This process slows down the performance of the parallel experiments by a constant amount of time.

1.5 Experimental results

This thesis will investigate the advantages and disadvantages of using a GPU as an alternative computation platform for CA simulations. It is thus important

Stellenbosch University https://scholar.sun.ac.za

to have access to experimental results. The framework must be able to provide experimental data such as computation time per time step, average computation time over a number of time steps, total accumulative computation time, and the number of frames that are drawn per second.

2 Design

Following the core requirements specified in Section 1, this section will analyze the core design of the framework and how it meets the requirements. An overview is given of the design of the framework, followed by the details of the individual components of the framework.

In order to add parallel algorithms to the framework, the CUDA software development kit will be used. Therefore, a programming language that supports CUDA is required to develop the framework. Currently, there are solutions for Fortran, C/C++, C#, and Python. For this thesis, the C/C++ programming language will be used, as it officially supports CUDA and incorporates the software paradigms discussed in Section 1.

2.1 Framework overview

Figure 3.1 provides an overview of the implementation of the framework. Each node in the diagram represents a class that is implemented in C++. Edges between nodes represent interaction between the different classes. In Figure 3.1 there are

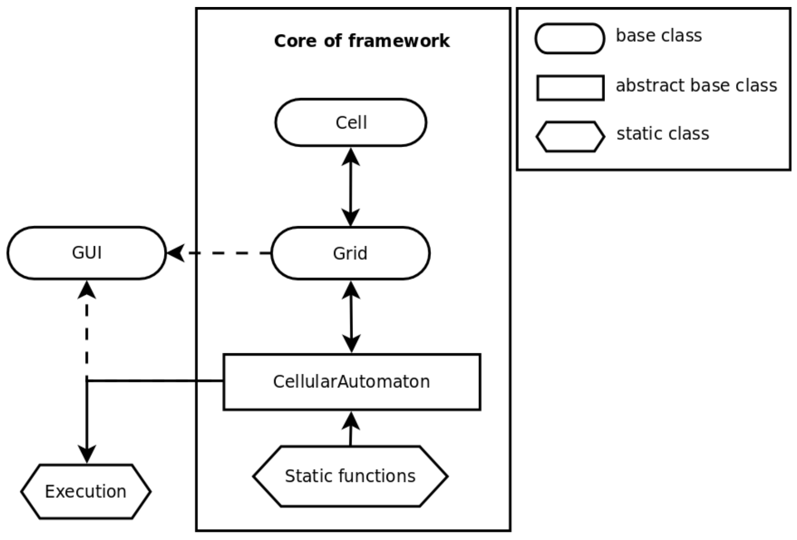


Figure 3.1: A basic overview of the proposed framework.

Stellenbosch University https://scholar.sun.ac.za

six primary classes, four of which make up the core of the CA framework, namely: `Cell`, `Grid`, `CellularAutomaton`, and `StaticFunctions`. The `GUI` class is used to draw a user interface, which will display the CA and render the generations of the CA. The `Execution` class starts the program and creates an instance of the GUI. A user can perform the experiments with the additional functionality provided by the GUI.

The following subsections will discuss the core classes listed above, in more detail.

2.2 CA core

The `Cell` class stores all the basic attributes and functions of a cell, as is shown in Figure 3.2. The basic attributes include dimensional values and the state value of

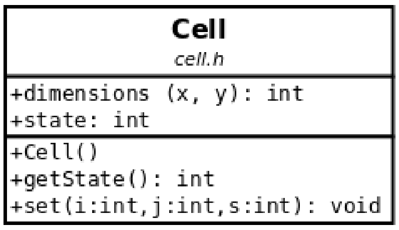


Figure 3.2: The `Cell` base class.

the cell. A standard getter function is included to return the state of the cell. A `set()` function is included and is primarily used for initializing each cell of the CA. It is important to note that all attributes that must be accessed when calculating the next state of a cell, must be made public, since CUDA does not allow function calls of objects during kernel executions.

Since extra attributes might be required for a cell, depending on the specific CA, the `Cell` class is used as a base class, and is extended to include additional attributes. To reiterate, any additional attributes that are used when calculating the next state of a cell, must be defined as public.

The `Grid` class stores a grid of cells for the CA and provides relevant utility functions; refer to Figure 3.3. The two `Cell*` attributes are initialized as arrays of type `Cell`; `grid` is used as the primary array of the CA cells. The array `grid_temp` is used as temporary storage, and is used (for certain CA such as Game of Life) when calculating the overall next state of the CA. The dimensions of the grid are stored in `dimx` and `dimy`, and are declared public to provide parallel algorithms with data (for example, it is used when mapping GPU threads to cells).
Three base utility functions `printGrid()`, `copyGrid()`, and `getGridStartIndex()` are provided. The `Grid` class is not abstract, but can be extended to provide addi-

Stellenbosch University https://scholar.sun.ac.za

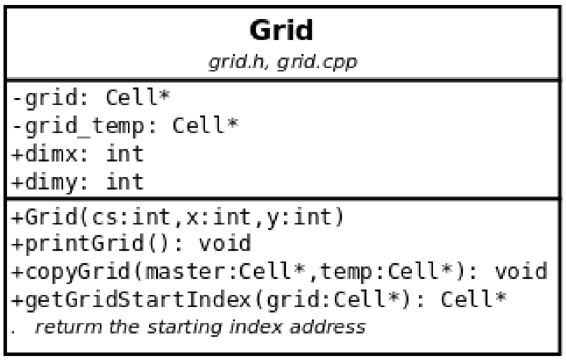

Grid
grid.h, grid.cpp
-grid: Cell*
-grid_temp: Cell*
+dimx: int
+dimy: int
+Grid(cs:int,x:int,y:int)
+printGrid(): void
+copyGrid(master:Cell*,temp:Cell*): void
+getGridStartIndex(grid:Cell*): Cell*
. returm the starting index address

Figure 3.3: The `Grid` base class.

tional attributes such as an extra dimension to the CA, or by providing additional utility functions.

The `CellularAutomaton` class combines the `Cell` and the `Grid` classes. In addition, the `CellularAutomaton` class provides the sequential and parallel algorithms wherein the rules of the specific CA are defined, and are used to calculate the next generation of the CA. Figure 3.4 expands on the base attributes and functions required for a standard CA. The attributes `csize`, `live`, and `dead` are used for

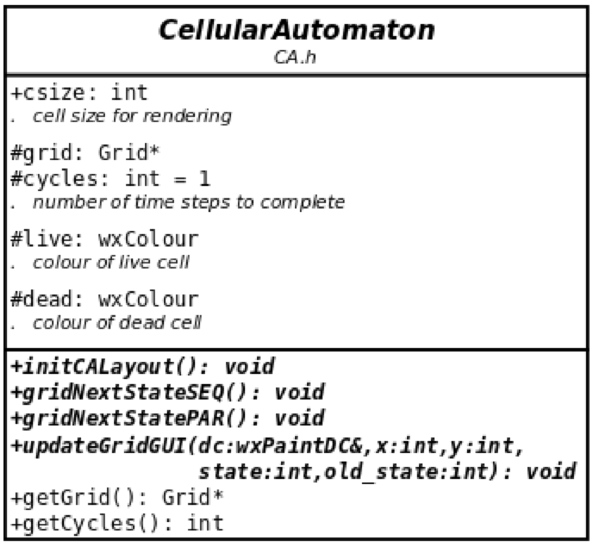


Figure 3.4: The `CellularAutomaton` abstract base class.

rendering, and refer to pixel space to occupy per cell, the colour of a *live* cell, and the colour of a *dead* cell, respectively. A pointer to the grid of cells is provided by the `grid` attribute.

Stellenbosch University https://scholar.sun.ac.za

The `CellularAutomaton` class declares four abstract functions, that are to be implemented according to a specific CA. The function `initCALayout()` provides a procedure which initializes the CA (setting the states and other relevant attributes of specific cells of the CA). The following two functions, `gridNextStateSEQ()` and `gridNextStatePAR()`, define the sequential and parallel algorithms to be executed when calculating the next generation of a CA. Finally, the function `updateGridGUI()` specifies how the CA must be rendered. Two utility functions `getGrid()` and `getCycles` return a handle to the grid of cells, and the number of generations to execute, respectively.

The `CellularAutomaton` class is defined as an abstract base class, and provides a programmer with a general basis upon which a new unique CA can be defined.

2.3 GUI

The `GUI` class acts as a static class and provides a platform for rendering the user interface of the framework. The `CellularAutomaton` class provides the `GUI` class with the function `updateGridGUI()`. The function call is made by the `GUI` class when rendering the next state of the particular CA.

The C++ GUI library, wxWidgets [63], is used for the framework to render the CA, as well as provide a GUI for user-system interaction. Since a modular approach is taken to implement the framework, other graphical drawing libraries for C/C++ can be used to render the CA and GUI (depending on the need of the programmer).

The GUI is also designed to print the time taken to calculate the next state change of the CA, along with the aggregated time to calculate x generations of the CA. This data is used for analysis of the particular algorithm used. Figure 3.5 gives a schematic overview of the GUI. Buttons to control the simulation of a CA are placed in the simulation control area. The CA is displayed in the CA simulation area, which shows the change in overall state of a CA. The printout area is used to print required data during the simulation of a CA.

2.4 Execution

The `Execution` class is used to initialize all procedures involved in setting up a newly defined CA, and to then start the simulation of the CA. It is essentially used as the main class from where execution starts, and provides necessary macros needed by the wxWidgets GUI library in order to transfer runtime control to the GUI.

Stellenbosch University https://scholar.sun.ac.za

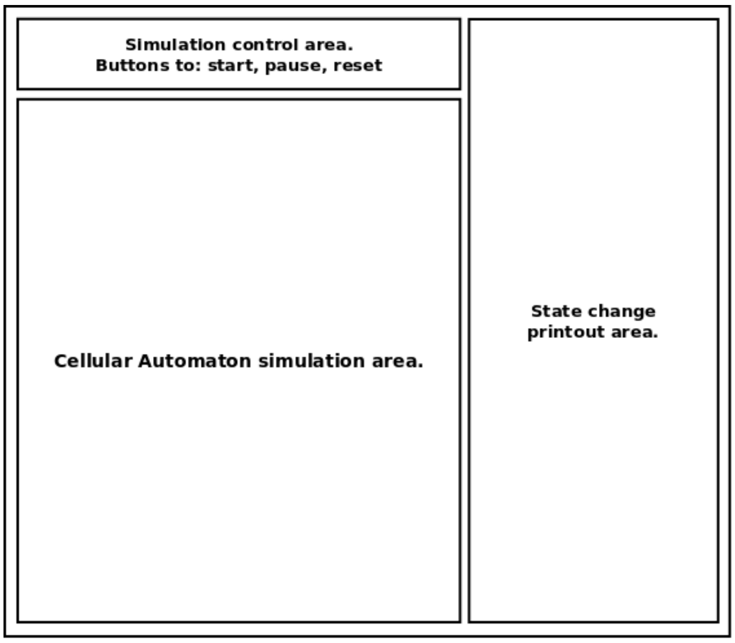

Simulation control area.
Buttons to: start, pause, reset
Cellular Automaton simulation area.
State change printout area.

Figure 3.5: A basic schematic overview of the GUI.

3 Implementation

In this section, specific aspects of the implementation phase are discussed. A closer look is taken at how CUDA is used for the parallel algorithms which apply the CA rules. Other aspects such as the GUI and problems that were encountered during the implementation of the framework, are briefly discussed.

3.1 CUDA

A brief overview of CUDA is given in Chapter 2, page 11, where the concepts of host-device interaction and data segmentation are discussed. This subsection expands on these ideas and explains their integration into the framework.

3.1.1 CUDA work structure

When solving problems using CUDA, a default set of procedures must be followed. Figure 3.6 gives a visual representation of the overall procedure. In Figure 3.6, the blue boxes represent instructions performed by the CPU, with the CPU being referred to as `SEQ`. The green boxes represent instructions performed by the GPU, with the GPU being referred to as `PAR`. Red boxes represent memory transfers between the CPU and the GPU.

After having defined the data set on which the work will be performed (a grid of

Stellenbosch University https://scholar.sun.ac.za

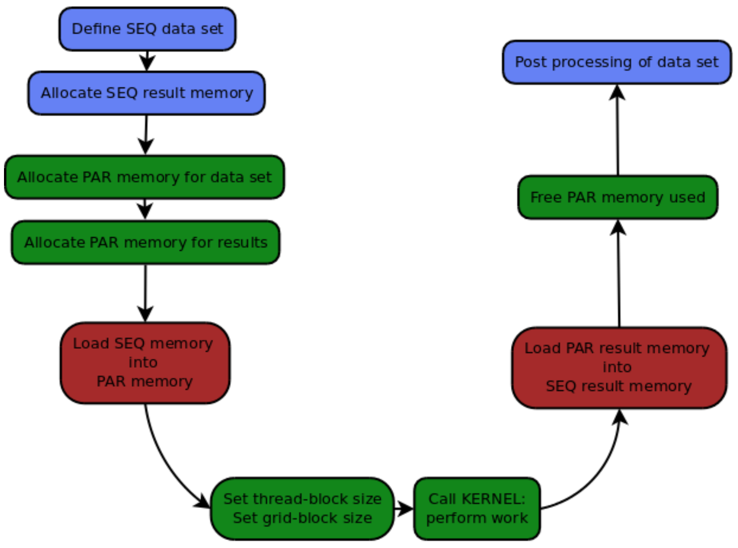

Define SEQ data set
Allocate SEQ result memory
Allocate PAR memory for data set
Allocate PAR memory for results
Load SEQ memory into PAR memory
Set thread-block size Set grid-block size
Call KERNEL: perform work
Load PAR result memory into SEQ result memory
Free PAR memory used
Post processing of data set

Figure 3.6: The set of procedures followed when solving a problem using CUDA.

cells for CA), an appropriate amount of memory must be allocated on the GPU and a pointer assigned to the memory. Memory, for storing the results, must also be allocated on both the GPU and in main memory, and pointers must also be assigned to the resultant memory. For certain CA, the results can replace the original data set and thus additional memory is not required.

To allocate memory on the GPU and to transfer data between main memory and GPU memory, the correct amount of memory must be calculated. The C unary operator `sizeof` calculates the size of the derived cell type used for the particular CA. This result is multiplied by the number of cells in the CA, to get the value of the total amount of memory that needs to be allocated and transfered.

For memory allocation on the GPU, the following CUDA built-in function is called:

- `cudaMalloc(void** devPtr, size_t size)`.

The address of the device memory pointer, as well as the amount of memory to allocate is provided as arguments. To transfer memory, the following CUDA built-in function is called:

- `cudaMemcpy(void* dst, void* src, size_t size, kind)`.

The first two arguments specify pointers to the destination and source memory locations, respectively. The third argument specifies the amount of data to transfer. The final argument specifies the kind of transfer to perform; either a transfer from the *host* to the *device*, or from the *device* to the *host*, and are specified as either:

- `cudaMemcpyHostToDevice`, or

Stellenbosch University https://scholar.sun.ac.za

- `cudaMemcpyDeviceToHost`.

After memory allocation, the original data set is loaded from main memory into the global memory of the GPU. The allocation and transfer of memory from main memory to the GPU, work in tandem, and have thus been combined into a single function which is part of the `StaticFunctions` class.

Next, the data set is segmented by setting the number of threads per thread-block and the dimensions of the grid of thread-blocks (Section 3.1.2). After these steps have been completed, the GPU kernel process which will perform the work, is called. A kernel process is called as one would call a normal C/C++ function, with an additional set of arguments provided as shown in the following example:

- `kernel_foo<<<grid_size, block_size>>>(arg1, arg2, ...)`.

The additional arguments are enclosed in the `<<< >>>` angle brackets, and provide the GPU with the dimensional information regarding the number of threads per thread-block and the number of thread-blocks it needs, when spawning threads to perform the work.

When the kernel process has finished, the processed data set in the GPU memory is loaded back to main memory, and all memory allocated on the GPU is freed using the CUDA built-in function

- `cudaFree(void* devPtr);`

The argument specifies a pointer to the allocated GPU memory used.

CUDA simplifies almost all the procedures discussed in this subsection, with the built-in functions listed. However, the procedure of data segmentation does require more input from the programmer than simply deciding how many threads are to be used when solving a problem. The following subsection discusses the process of data-to-thread assignment in detail.

3.1.2 Data segmentation

Data segmentation involves the utilization of processing resources on the GPU. The CUDA enabled GPU architecture is made up of a number of multi-threaded Streaming Multiprocessors. Each Streaming Multiprocessor (SM) is designed to execute hundreds of threads concurrently, and in order to do so, it uses a SIMT (Single Instruction, Multiple-Thread) architecture [37]. The number of threads that are processed simultaneously is equal to the total number of CUDA cores that a GPU has. It is important for a programmer to maximize the use of processing resources available, as maximum resource utilization generally increases the overall performance gained [30, 37].

The threads performing the work are sectioned into one, two, or three-dimensional thread-blocks; all thread-blocks are the same size and a thread-block is limited to

Stellenbosch University https://scholar.sun.ac.za

a maximum of 1024 threads. During the execution of a parallel code segment, each thread-block is assigned to an SM, and a programmer should ideally create at least as many thread-blocks as SMs [5, 37].

When segmenting a data set, the constraint on the maximum number of threads per thread-block must be taken into account. The general approach for data segmentation is to have a number of thread-blocks, each assigned to a subset of the data set. Since each thread-block is assigned to an SM, it is advantageous to create at least as many thread-blocks as the number of SMs. This allows more data to be processed concurrently, depending on factors such as global memory access between threads and optimal instruction usage [5, 30, 37].

Thread-blocks combined are known as a grid, and the dimensions of the grid are defined by the number of thread-blocks in every dimension. Figure 3.7 shows an

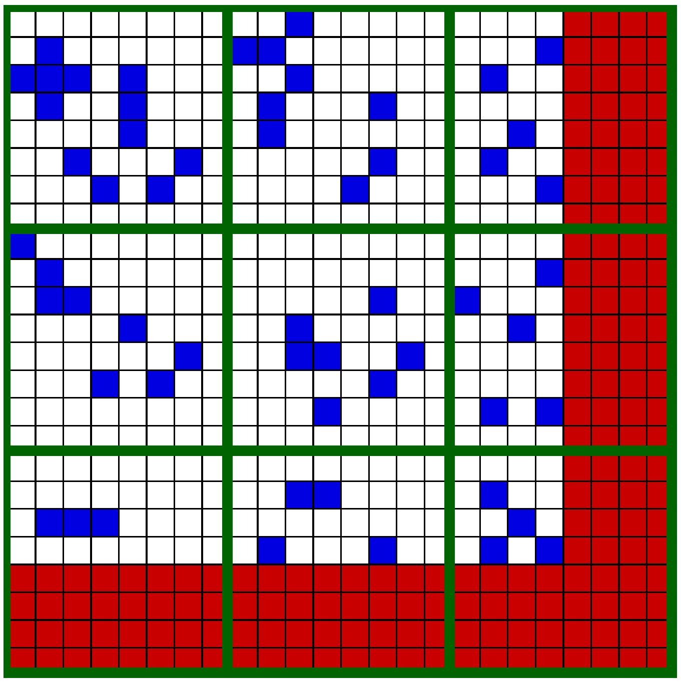


Figure 3.7: A twenty by twenty CA (blue and white blocks), segmented into a grid of thread-blocks. Each thread-block is made up of an eight by eight block of threads.

example of a data set divided into a grid of thread-blocks. The thread-blocks are marked in green. Red cells in the figure represent threads to which work cannot be assigned as there are less cells in the CA than there are threads. Each thread-block is assigned to one of the SMs which spawns 8^2 threads. An SM then maps each of its threads to a different cell in the in the CA. Thus, each SM will process a maximum of $8^2 = 64$ cells.

Stellenbosch University https://scholar.sun.ac.za

One way of segmenting the data of a CA (the cells to be processed), is to assign a sub-grid of cells to a single thread, also known as grid-per-thread assignment [30]. The number of sub-grids to be processed, is equal to the number of SMs that the GPU has. By following this approach, only one n^{th} of the available calculation power per SM is being utilized, where n is the number of CUDA cores that each SM has. This is because only one thread is spawned per SM to process the data, and therefore an SM only uses one CUDA core to execute the thread. If the data set is small, the impact on performance will not be significant, as each thread will only need to processes a few cells. However, as the size of the CA grid increases, each thread will have to process a larger number of cells, which inevitably will cause the overall performance to diminish.

An alternative approach is a row-per-thread assignment. In this scenario, an entire row of cells is assigned to a single thread, where the number of threads to utilize is equal to the number of rows in the CA. Thus, a single SM will process up to 1024 rows worth of cells, where the number of cells that are simultaneously processed is equal to the number of CUDA cores per SM. Since any CUDA enabled GPU has more CUDA cores per SM than SMs per GPU [37], one is able to process more cells simultaneously than with grid-per-thread assignment. However, only one SM per 1024 rows in the CA is being utilized. To increase the number of threads that are concurrently processed in the row-per-thread assignment, one can divide the number of rows in the CA by the number of SMs that the GPU has, and then assign a subset of rows to each SM. For example, If a GPU has two SMs, and 192 CUDA cores per SM, and has to process a square CA of 1000 rows by 1000 columns, the first row-per-thread segmentation method will assign all 1000 rows of cells to a single SM, and the SM will process 192 rows of cells simultaneously. The second row-per-thread segmentation method will assign 500 rows of cells to each SM, and each SM will process 192 rows simultaneously, thereby doubling the number of rows processed.

For both the grid-per-thread and row-per-thread assignment methods, each thread calculates the state of a number of cells. A final approach to consider (and which is generally adopted for problems solved with a GPU) is to assign a single cell to a single thread. This method, known as cell-per-thread assignment, generally tends to use all processing resources of the GPU, depending on the size of the data set as well as the dimensions specified for the thread-blocks. All the data segmentation methods will be analyzed in Chapter 4.

The dimensions of a thread-block and grid are defined using the CUDA data type `dim3`. For example, the size of a two-dimensional thread-block and grid of thread-blocks for a two-dimensional CA are respectively defined as:

- `const dim3 block_size(k, k, 1)`, and
- `const dim3 grid_size (dimx/k, dimy/k, 1)`.

Stellenbosch University https://scholar.sun.ac.za

Here, `k` is a constant value in the range 1–32 (so as not to exceed the threshold of 1024 threads per thread-block), and `dimx` and `dimy` are the `x` and `y` dimensions of the two-dimensional CA grid.

3.1.3 CUDA kernel functions

The C/C++ functions that are executed by the GPU are known as kernel functions. Kernel functions are preceded by either the `__global__` or `__device__` macros. The `__global__` kernel function must be declared as void, as it cannot return a value or reference after execution. These kernel functions are called from a standard C/C++ function and control is passed to the GPU. The `__device__` kernel function can return a value or reference and is used primarily as utility functions. A `__device__` kernel function can be called from another `__device__` kernel function or from a `__global__` kernel function.

Both kernel function types are coded as normal C/C++ function. However, object method calls are not permitted in kernel functions. Therefore, any object argument passed to a kernel function must be set up in such a way that attributes, that are to be changed, can be accessed without the need for object function calls.

As with standard C/C++ functions, a kernel function needs access to the data set on which the algorithm, contained in the kernel function, must be performed. The pointer assigned to the memory location of the data set must be given as an argument, and unless the same memory will be altered, a pointer to the resultant memory location must also be provided. Any specific parameters required to perform the algorithm must also be provided.

In order to map threads to the data set, `__global__` kernel functions have access to the following built-in CUDA kernel indexing parameters:

- `blockIdx`,
- `blockDim`, and
- `threadIdx`.

These indexing parameters adopt the dimensional information specified in the `<<< >>>` angle brackets, during a kernel function call. Therefore, depending on how thread-blocks have been set up, the parameter `blockIdx` will store the X, Y, and Z index of the thread-block to which a thread belongs. The same applies to the `threadIdx` index parameter, which stores the thread index information of the specific thread being executed. The index parameter, `blockDim`, stores the dimensional information of a thread-block.

The framework developed for this thesis stores the data set as an array of cells. To index into a two-dimensional CA in the `__global__` kernel function, one would need the X and Y indices. The X index can be calculated as follows:

Stellenbosch University https://scholar.sun.ac.za

- `x = blockIdx.x * blockDim.x + threadIdx.x`

(the Y index is calculated similarly). Since the X index is a column offset into the Y'th row, the data of a specific cell at the index `(x,y)`, is retrieved as follows:

- `data_in[y * dimx + x]`,

where `dimx` is the number of cells per row in the CA.

3.2 Issues encountered

In this subsection, issues that were encountered during the implementation of the framework are discussed.

3.2.1 Independent C++ GUI integration

Visualization of a CA is an important aspect for any CA simulation. Since a standard GUI library is not part of C/C++, the choice of how to integrate the chosen GUI library into the framework was an important factor.

The original design for the framework was to include `wxColour` objects in the `CellularAutomaton` class, and to allow for the inclusion of additional `wxColour` objects. The reason was to allow the user to customize the visual simulation of the specific CA implemented with the framework. This approach does however restrict a user from using a different GUI library with the framework, or enforces the inclusion of the wxWidgets libraries when using the framework.

A decision was made to remove `wxColour` objects from the `CellularAutomaton` class and to create default `wxColour` objects in the `GUI` class. As a result, a set of default colours were chosen to represent the different states that a cell can have for the CA that were implemented with the framework.

This decision also removes the restriction that forces a programmer to use the GUI library that was chosen for this thesis. This is useful for users who want to use the framework along with a different GUI library or with a 3D rendering library, such as OpenGL.

3.2.2 CUDA CA algorithms

The difference between sequential and parallel algorithms essentially comes down to data segmentation. For a sequential algorithm, all work is assigned to one process, whereas for algorithms implemented with CUDA, data segmentation adds a new dimension to the overall implementation of the rule application function.

Since the effectiveness of different data segmentation methods is investigated for this thesis, overlapping code appeared when these methods were first implemented. By porting some sections of the algorithms to `__device__` kernel functions, a lot of redundant code was removed.

Stellenbosch University https://scholar.sun.ac.za

The use of `__device__` kernel functions have been applied to the process of calculating the neighbourhood of a cell, and rule application. By separating neighbourhood calculation from the rest of the algorithm, any future implemented CA which uses the same kind of neighbourhood, can make use of the same static function specifically designed to calculate the neighbourhood in question. Since different data segmentation methods are used for each CA implemented for this thesis, the part of the algorithm which applies the rules of the CA has been moved to a static function, which also significantly reduces redundant code.

3.2.3 Representing the grid data structure

The primary purpose of the `Grid` class is to store an array of cells which represents the grid of the CA. Originally, this data structure was represented with a two-dimensional matrix, `Cell[][]`. This approach resulted in two issues. The first issue was that this method imposed a limitation on the dimensions of the grid; it could only represent a one or two-dimensional grid of cells. The second issue involves defining functions that must use this matrix, since in C/C++, a matrix must have a predefined bound for the second dimension, when the function is first defined. For example, a function that receives a matrix as parameter, must be defined as:

- `void foo(Cell grid[][x])`,

where the value of `x` must be defined as a constant or must be replaced with an integer value. Because of this limitation, a programmer must change the second dimensional value `x`, in the code.

As these restrictions made the use of the standard C/C++ matrix difficult, a data structure using the standard C++ vector class was then used. That is, a matrix is a vector of vectors. However, the total size of the data structure was increased since each column of cells is represented by a vector object. This caused the copy function to reduce the performance of the sequential algorithm, and apart from this issue, the limitation of the dimensions of the grid was still present.

Finally, to take care of the performance decrease and the dimensional limitation of the grid, the grid was changed to be a `Cell` pointer. By using a `Cell` pointer, the grid is effectively defined as an array. Indexing into the array was then achieved by standard indexing techniques. If the `Cell` array, `grid[]`, represents a grid of N dimensions, indexing into `grid[]` is achieved by

- `grid[`$n_1 + (n_2 * d_1) + (n_3 * d_2 * d_1) + \ldots + (n_N * d_{N-1} * d_{N-2} * \ldots * d_2 * d_1)$`]`.

In this equation, n_x represents the index into the x^{th} dimension, and d_x represents the length of the grid in the x^{th} dimension.

It is important to note that changing from a matrix representation to a array representation will not change the order in which cells are stored in memory. Thus,

Stellenbosch University https://scholar.sun.ac.za

the C++ row-major order is retained. Also note that since data is stored in a row-major order in both the main memory and GPU global memory, an issue of data non-locality does occur when processing the neighbourhood information of a cell. However, the data non-locality issue will occur for both the CPU and the GPU when processing a cell [5], and the time required to read all neighbouring cells into cache memory will be consistent on both the CPU and GPU. Therefore, data non-locality is ignored during experimentation.

4 Fulfillment of requirements

In this section the design and implementation of the framework will be assessed, by considering how the requirements (as discussed in Section 1) are met.

4.1 Modularity and abstraction

The core of the framework has been designed to be modular, as discussed in Section 2.2. Abstraction is applied to the `CellularAutomaton` base class to simplify the coding process involved with execution and visualization of a CA. When creating a new CA, the user simply extends the `Cell` class as needed, and adds additional cell attributes. The derived `Cell` class is then passed to the `Grid` class using a special constructor. After setting up the `Grid`, it is passed to the default `CellularAutomaton` constructor, used by all derived `CellularAutomaton` classes. Using a modular approach simplifies the construction of new CA, and helps with applying changes to other modules such as the `GUI` class and `StaticFunctions` class.

4.2 CA algorithms

In order to reduce redundant code in algorithms for CA which use the same neighbourhoods, functions that calculate the relevant neighbourhood of a particular cell, are separated from the algorithm which calculates the next state of the CA. These functions are added to the `StaticFunctions` class. To make use of implemented functions, the `StaticFunctions` header file is included in the `CellularAutomaton` base class.

All unique CA that are derived from the `CellularAutomaton` base class, set up the CA rule application algorithms by implementing the

- `gridNextStateSEQ()`, and
- `gridNextStatePAR()`

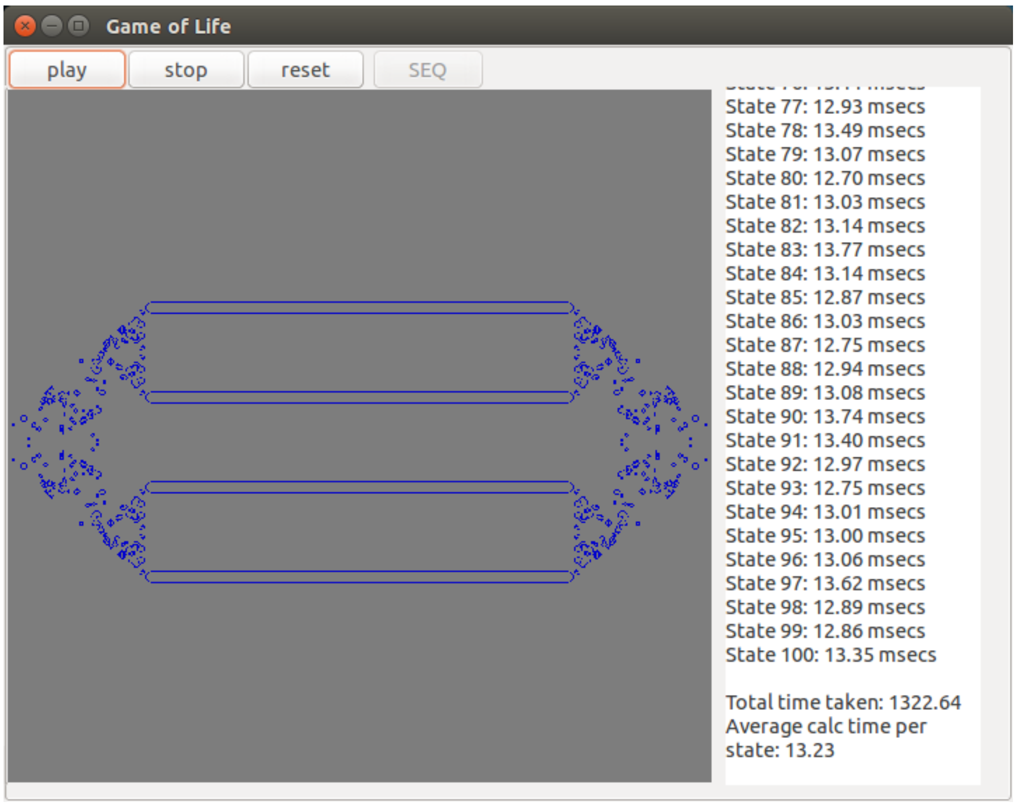
functions. For this thesis, these functions are called from the `GUI` class. Finally, in order to determine the calculation time of the algorithms, the C++ standard

Stellenbosch University https://scholar.sun.ac.za

`chrono` library is used for sequential algorithms. For parallel algorithms, a helper utility class `GpuTimer` is used.

4.3 Visualization

For this thesis the chosen GUI library, wxWidgets, is sufficient for runtime control of CA simulations, displaying state change time-lapses, and visualizing the CA. Figure 3.8 gives an example of the GUI implementation. The top bar of the GUI



```
State 77: 12.93 msecs
State 78: 13.49 msecs
State 79: 13.07 msecs
State 80: 12.70 msecs
State 81: 13.03 msecs
State 82: 13.14 msecs
State 83: 13.77 msecs
State 84: 13.14 msecs
State 85: 12.87 msecs
State 86: 13.03 msecs
State 87: 12.75 msecs
State 88: 12.94 msecs
State 89: 13.08 msecs
State 90: 13.74 msecs
State 91: 13.40 msecs
State 92: 12.97 msecs
State 93: 12.75 msecs
State 94: 13.01 msecs
State 95: 13.00 msecs
State 96: 13.06 msecs
State 97: 13.62 msecs
State 98: 12.89 msecs
State 99: 12.86 msecs
State 100: 13.35 msecs

Total time taken: 1322.64
Average calc time per
state: 13.23
```

Figure 3.8: An example of the GUI, created with wxWidgets.

contains buttons that the user can use to interact with CA simulation. These include starting, stopping, and resetting the CA, and selecting which algorithm to use (sequential or parallel). Below the buttons, to the left is the display area of the current CA generation. To the right is a print area where the calculation time of each generation is printed. Once the simulation has finished, the total time taken as well as the average calculation time per generation, is printed.

The next chapter will analyze specific CA implemented with the framework discussed in this chapter.

Stellenbosch University https://scholar.sun.ac.za

Chapter 4

Experiments and results

1 Overview

With the CA framework in place, specific CA implemented using the framework are discussed in detail in this chapter. For each CA, a sequential rule application function and parallel rule functions, based an the data segmentation methods (discussed in Section 3.1.2 on page 22), are implemented. The time taken to calculate a number of generations for different CA grid sizes is measured and is analyzed for each implementation. The CA implemented with the framework are:

- Conway's Game of Life [14];
- clay deformation based on free form shape modeling, proposed by Arata et al [3]; and
- ant clustering as proposed by Lumer et al [31].

These three CA have different state calculation functions that can produce vastly different measurable performance results, when performed on the same grid sizes. Game of Life is the most basic of the three CA proposed, as its state calculation function only consists of conditional checks (`if` statements) and integer arithmetic.

The clay deformation CA, requires floating point arithmetic for its state change function. These floating point arithmetic operations are performed in a so-called stabilization function, which is repeated until the clay structure is stable.

The ant clustering CA has a complex rule application function, which relies on random number generation. This CA also has a different state update function to the previous two CA, where cells to which the rules are applied, can shift to random neighbouring locations in the CA grid. For a parallel algorithm, which updates many cells simultaneously, collisions can occur where two or more cells are moved to the same location. As a result, cells are updated conditionally to avoid collisions.

Stellenbosch University https://scholar.sun.ac.za

1.1 Hardware used for experiments

For this thesis, two computers (A and B) will be used; thus two CPU and GPU configurations are used to gather the experimental results. Computer A is less powerful than Computer B. These computers have been selected to check for consistency in the experimental data gathered, to check how the different GPUs perform against each other, and how the slower GPU performs against the faster CPU.

For the CPUs and the GPUs, the clock speeds are provided in Table 4.1 and Table 4.2. Additionally for the GPUs, the SM count, CUDA cores, memory bus width and bandwidth are provided in Table 4.2.

	Computer A	**Computer B**
CPU model	i7-3630QM	i5-4690
CPU clock speed	2.40 GHz	3.50 GHz

Table 4.1: Intel CPUs used for sequential experiments.

	Computer A	**Computer B**
GPU model	GeForce GT 650M	GeForce GTX 780
GPU core clock speed	950 MHz	954 MHz
SM count	2	12
CUDA cores	384 (192 per SM)	2304 (192 per SM)
Memory bus width	128 bit	384 bit
Memory bandwidth	80 GB/s	288.4 GB/s

Table 4.2: NVIDIA GPUs used for parallel experiments.

In terms of the CPUs used, the CPU of Computer A is 1.1 GHz slower than the CPU of Computer B, and as a results we expect Computer B to produce better time performance data for all sequential algorithms. In terms of the GPUs used, Computer A and Computer B have GPUs clocked at nearly the same clock speed (a 4 MHz difference). However, Computer A only has two SMs in contrast with the 12 SMs of Computer B. Thus we expect Computer B to perform better than Computer A, when there is an increase in the number of additional cores used.

Another noticeable difference between the GPUs of Computer A and Computer B is the memory bus width. Computer B has a bus width which is three times wider than Computer A; therefore, more data can be copied from main memory to the GPU global memory of Computer B, per second (refer to the memory bandwidth in Table 4.2). Since both Computer A and Computer B use DDR3-1600 MHz dual channel memory controllers, we expect memory bottlenecks to occur as a result of the memory transfer speed offered by the GPUs.

Stellenbosch University https://scholar.sun.ac.za

2 Game of Life

The first CA that will be analyzed is Conway's classical Game of Life [14]. Game of Life is a simple two-dimensional CA used to simulate life, death, and survival through the evolution of an initial pattern of live cells.

A Game of Life initial state configuration can produce one of three conditions after having evolved over a number of generations [14, 45]:

- extinction, where all cells have died out;
- stability, where all live cells stay alive and do not change state anymore; and
- oscillation, where live cells enter into an endless cycle.

2.1 Rules

The rules of Game of Life are based on the simulation of life, death, and survival, by checking the neighbourhood of a cell for being overcrowded (too many live cells in the neighbourhood), having a lack of life (not enough live cells in the neighbourhood), or having an appropriate number of live cells in its neighbourhood to sustain the life of the cell into the next generation. Cells are either alive (have a state of '1') or dead (have a state of '0'). The rules of Game of Life are as follows:

1. A live cell will die if it has less than two live neighbours (under-population).
2. A live cell will die if it has more than three live neighbours (overcrowding).
3. A live cell will stay alive in the next generation if it has two or three live neighbours.
4. A dead cell will become alive if it has exactly three live neighbours. Else, it will stay dead.

The type of neighbourhood used for the Game of Life CA is a Moore neighbourhood with a one cell radius [45], which, for a two-dimensional CA, includes all four orthogonally adjacent and all four diagonally adjacent neighbours (see Figure 4.1).

2.2 State calculation analysis

In order to calculate the next state of a cell, eight conditional checks must be performed. A conditional check is performed on each neighbouring cell, of the cell in question. During the conditional check, the state of a cell is checked to determine whether the cell is alive or dead. An integer counter, which counts the number of live neighbouring cells of the current cell is incremented for each live neighbour.

Stellenbosch University https://scholar.sun.ac.za

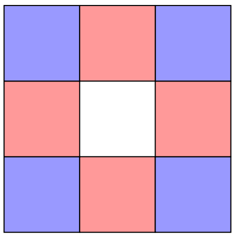


Figure 4.1: Two-dimensional Moore neighbourhood, with a one cell radius. Red neighbours are orthogonally adjacent. Blue neighbours are diagonally adjacent.

Since the CA grid is stored as an array, the state look-up is performed in constant time by specifying the index of a neighbouring cell into the array.

For each cell in the CA, eight array indexing operations (along with eight conditional checks) are performed. Depending on the number of live cells in the neighbourhood of a cell, a maximum of eight additional integer arithmetic operations are performed, to increment the live neighbour counter. Finally, a conditional check is performed depending on the current state of the cell, to determine what its state will be in the next generation.

For an $M \times N$ sized grid, where M is the number of rows and N the number of columns, the operations described will have to be performed $M \times N$ times. Thus the work complexity of the Game of Life CA is $O(MN)$. All operations are performed in constant time, and the time complexity is $O(MN)$. For convenience, without the loss of generality, a square grid CA is used in all the experiments. Thus, $M = N$ and the time complexity is $O(N^2)$.

2.3 Experimental setup

All the Game of Life CA experiments are performed for the grid sizes 250^2, 500^2, 750^2, 1000^2 and 1250^2, to test the scalability and to investigate the time complexity of of each implementation. For each grid size, 100 generations of the CA is simulated in order to reduce the number of outliers (generations which took above the average time to calculate). The total time to calculate 100 generations is measured in milliseconds. This process is repeated ten times to validate the calculation times measured.

2.4 Sequential implementation

In this subsection the sequential implementation of the Game of Life CA is considered.

Stellenbosch University https://scholar.sun.ac.za

2.4.1 Sequential code extracts

The reader may refer to Listing B.1 for details on the sequential implementation. The rule application algorithm, as discussed in Section 2.2 requires the calculation of the number of live cells for each cell in the CA. This calculation is performed with a separate function and provided in Listing B.2.

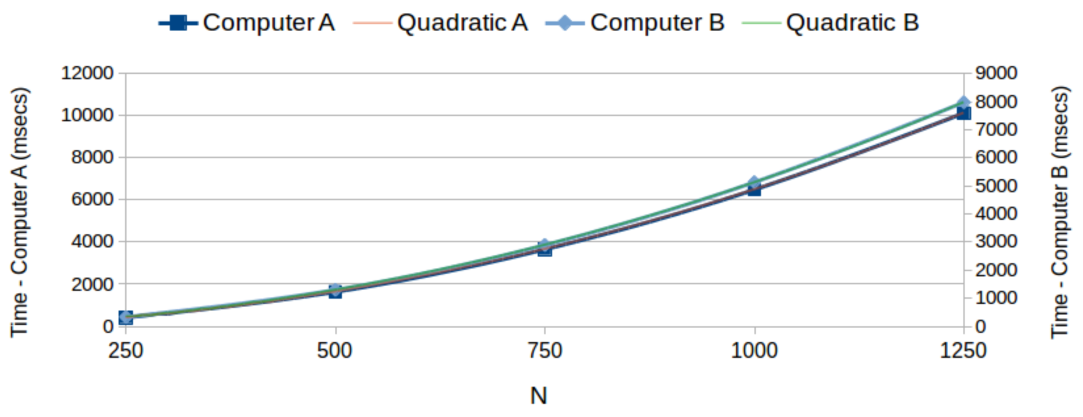
2.4.2 Sequential experiment results

Table 4.3 shows the average calculation times for each grid size and the average number of CA generations calculated per second, for both Computer A and Computer B. Figure 4.2 shows the calculation time series.

Table 4.3: Results for Game of Life sequential experiments.

	Calculation time (msecs)		Generations per second	
Grid size	**Computer A**	**Computer B**	**Computer A**	**Computer B**
250^2	414	330	241	303
500^2	1638	1298	61	77
750^2	3656	2895	27	35
1000^2	6492	5134	15	19
1250^2	10126	7976	10	13

Figure 4.2: Game of Life sequential calculation time with quadratic trend lines.



Computer A takes longer to calculate 100 generations, which also leads to fewer generations calculated per second, because Computer A has a slower clock speed than Computer B. The data indicates a quadratic time increase between $N = 250$

Stellenbosch University https://scholar.sun.ac.za

to $N = 500$, and $N = 500$ to $N = 1000$ as well as a quarter of the average number of generations calculated per second, which corresponds to the time complexity of $O(N^2)$ (refer to Table 4.4). When comparing the calculation time results with

Table 4.4: Expected results according to the time complexity of the algorithm, compared to actual data for Computer A and Computer B.

	Time increase constant		**Generation decrease constant**	
	Computer A	**Computer B**	**Computer A**	**Computer B**
Expected	4		0.25	
250→500	3.95	3.94	0.25	0.25
500→1000	3.96	3.95	0.25	0.25

the number of cells for the value of N, a correlation of 0.999999403 between Computer A and the number of cells is found, and a correlation of 0.999993957 between Computer B and the number of cells. Assuming it takes time t to calculate the state of one cell, and since cells are processed sequentially, it will take time N^2t to process all the cells for one generation. Thus, the correlation indicated above between the complexity of $O(N^2)$ and the calculation time results are as expected.

2.5 Parallel implementation

In this subsection the parallel implementations for each of the data segmentation methods used for the Game of Life CA is considered.

2.5.1 Game of Life parallel code extracts

As discussed in Section 3.1.2 of Chapter 3, there are four data segmentation methods that will be used. The first method divides the CA grid into sub-grids equal to the number of SMs a GPU has. Each sub-grid is then assigned to one thread, executed by on one of the SMs. Listing B.3 provides the algorithm used to perform the grid-per-thread rule application for Game of Life.

The next two data segmentation methods are based on row-per-thread assignment, and the algorithm to apply the rules is given in Listing B.4. For the first method of row-per-thread assignment, one thread-block is created for every 1024 rows of cells. If there are more than 1024 rows in the CA grid, an additional SM will be used to process the next 1024 rows of cells. The second method of row-per-thread assignment divides the number of rows in the CA grid, by the number of SMs in the GPU. The difference between the first and second method of row-per-thread assignment is in the dimension of the thread-blocks used. The thread-block dimension is set when providing the grid and thread-block size parameters in the angle brackets. For the first method of row-per-thread assignment the angle brackets are

Stellenbosch University https://scholar.sun.ac.za

set as `<<<dimy/tbm+1, tbm>>>`, where `tbm` is equal to the maximum number of threads allowed per thread-block. For the second method of row-per-thread assignment the angle brackets are set as `<<<smc, rptc>>>`, where `smc` is equal to the number of SMs of the particular GPU executing the algorithm, and `rptc` is equal to the rows assigned to each SM.

The final data segmentation method is based on cell-per-thread assignment, and the algorithm to apply the rules is given in Listing B.5. The assignment of a thread to a specific cell is performed by determining both the row and column indices of the cell, relevant to the thread and the thread-block to which the thread belongs. The thread-blocks are made up of $k \times k$ threads, where k is a value between 1 and 32. We have chosen $k = 16$, which is big enough to reduce the number of thread-blocks assigned to each SM, and small enough to reduce the thread-to-core assignment bottleneck per SM. Each thread-block is assigned to a $k \times k$ sub-grid.

All three functions given (for the different data segmentation methods) in Listings B.3, B.4, and B.5 use the parallel utility functions `N_Moore_GPU()` and `GoL_Rules_GPU()`. `N_Moore_GPU()` returns the number of live cells in the neighbourhood of a specific cell, identical to Listing B.2. `GoL_Rules_GPU` applies the rules of Game of Life to a cell, by referring to the current state of the cell and the number of live neighbours it has, as is shown in Listing B.6.

2.5.2 Parallel experiment results

Table 4.5 shows the data gathered for the grid-per-thread data segmentation method. Figure 4.3 shows the calculation time series.

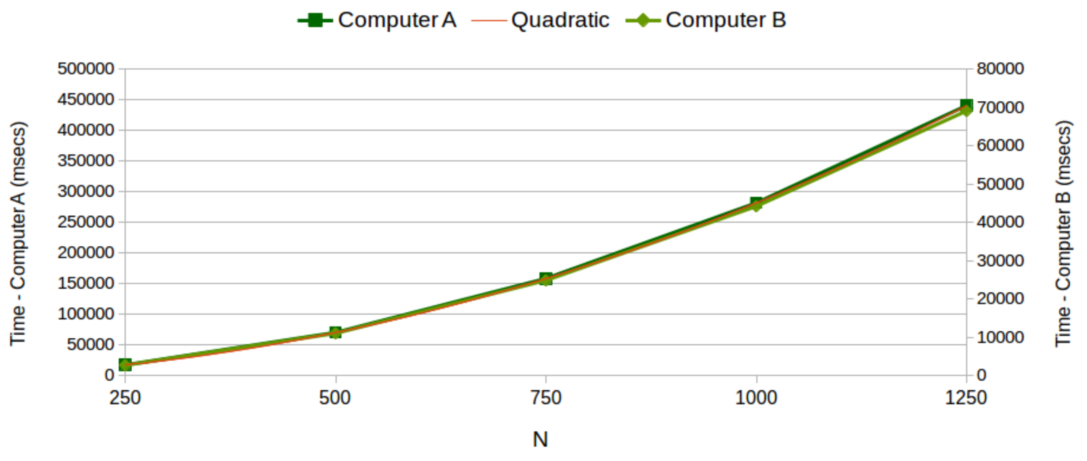
Table 4.5: Results for Game of Life parallel grid-per-thread segmentation method experiments.

	Calculation time (msecs)		States per second	
Grid size	**Computer A**	**Computer B**	**Computer A**	**Computer B**
250^2	17658	2755	5.66	36
500^2	70510	11068	1.42	9.03
750^2	158562	24898	0.63	4.02
1000^2	281953	44243	0.35	2.26
1250^2	440756	69135	0.23	1.45

For the grid-per-thread segmentation method the data is equally divided between the number of SMs. Each SM has a workload of $\frac{1}{S_c}N^2$, where S_c is the number of SMs of a GPU. Each SM processes one cell at a time (since only one thread is started to process the work), taking time t. Thus each SM takes time $\frac{1}{2}N^2 t$ to process its workload. A direct correlation between the N^2 growth in the work load

Stellenbosch University https://scholar.sun.ac.za

Figure 4.3: Game of Life parallel grid-per-thread method: calculation time with a quadratic trend line.



and the time taken to process all the cells is expected (refer to Figure 4.3). The data shows a 0.9999997181 correlation between the calculation time of Computer A and the work load increase, and a 0.9999997775 correlation between the calculation time of Computer B and the work load increase.

Table 4.6 shows the data gathered for the first row-per-thread data segmentation method. Figure 4.4 shows the calculation time series.

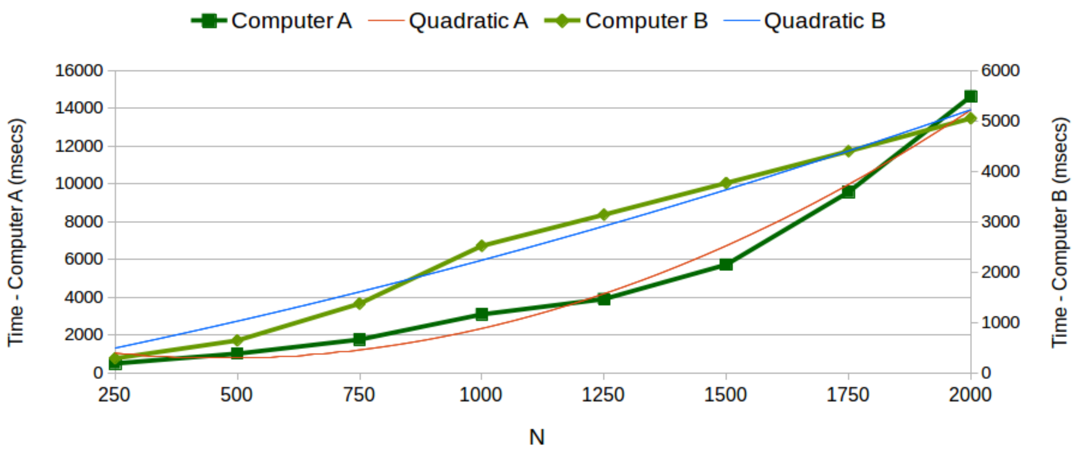
Table 4.6: Results for Game of Life parallel row-per-thread segmentation method one experiments.

	Calculation time (msecs)		**States per second**	
Grid size	**Computer A**	**Computer B**	**Computer A**	**Computer B**
250^2	483	283	207	353
500^2	1005	638	100	157
750^2	1747	1368	57	73
1000^2	3083	2514	32	40
1250^2	3891	3135	26	32

The first row-per-thread segmentation method assigns a maximum of 1024 rows of cells to each SM; each row containing N cells. For $N \leq 1024$ the SM to which the work is assigned must process N^2 cells. For $N > 1024$ the next available SM will process the remaining r rows, where $r \leq 1024$. In the case of $N = 2048$, two SMs will be used, each one having a workload of $1024 \times N$ cells. In this case $1024 = \frac{1}{2}N$,

Stellenbosch University https://scholar.sun.ac.za

Figure 4.4: Game of Life parallel row-per-thread method one: calculation time with quadratic trend lines.



and each SM thus has a workload of $\frac{1}{2}N^2$. In general, depending on the number of SMs of the GPU, for every $N = 1024k$, where $k = 1, 2, 3, \ldots, S_c$, each SM will have a workload equal to $\frac{1}{k}N^2$.

For experiments performed using the first row-per-thread segmentation method, only the experiment for $N = 1250$ utilizes a second SM to process the remaining rows. Since both SMs will start processing their workloads simultaneously, a decrease in the time needed to calculate the 1250^2 grid size is expected, relative to the time-increase for grid sizes: 250^2, 500^2, 750^2, and 1000^2. Figure 4.4 shows this trend; there is a noticeable difference between the asymptotic growth from $N = 750$ to $N = 1000$ to $N = 1000$ to $N = 1500$.

Analyzing the time complexity of the first row-per-thread segmentation method, one has to take into account that each SM has 192 cores, which process work simultaneously. Once processing starts, an SM assigns the first 192 rows of cells to its cores, each core processing one cell at a time. If it takes time t to process 192 cells simultaneously, it will take $\frac{1}{192}N^2t$ to process N^2 cells, for $N < 1024$. In general it will take time equivalent to $\frac{1}{192S_c}N^2t$, for $N > 1024$. Referring to Figure 4.4, a relative correlation between the computation times and the quadratic trend lines is noticeable. When an additional SM is utilized, the growth in computation time decreases slightly since an additional 192 cells are being processed. However, as N increases, there are more cells per row to process. As N reaches the next multiple k of 1024, each SM takes time $\frac{1}{192}1024^2kt + \frac{1}{192}N^2 - 1024kt$ or $\frac{1}{192}(N^2 + 1024k(1023))t$ to process its workload. The dominating factor in this time complexity function is N^2, and therefore as $N \to \infty$, a stable quadratic increase in the computation time is expected.

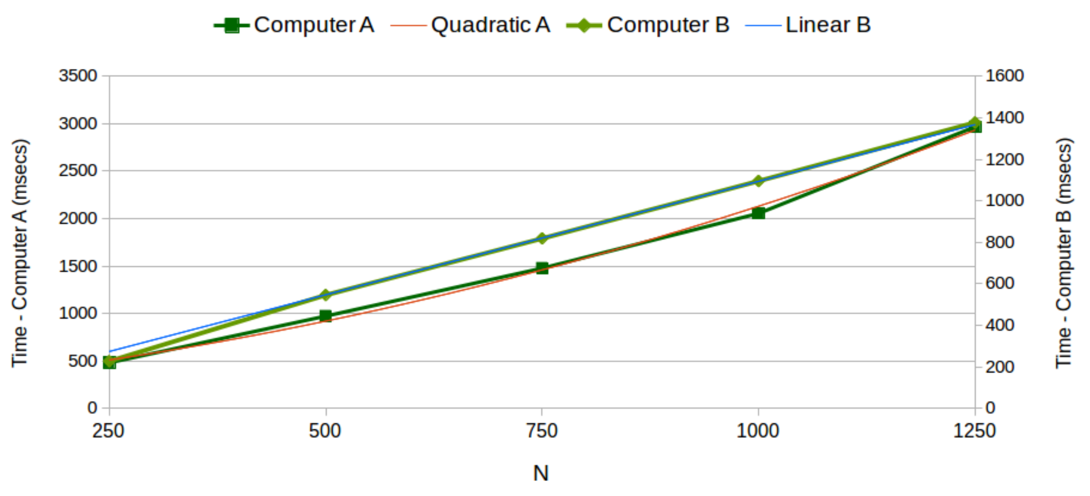
Stellenbosch University https://scholar.sun.ac.za

Table 4.7 shows the data gathered for the second row-per-thread data segmentation method. Figure 4.5 shows the calculation time series.

Table 4.7: Results for Game of Life parallel row-per-thread segmentation method two experiments.

	Calculation time (msecs)		States per second	
Grid size	**Computer A**	**Computer B**	**Computer A**	**Computer B**
250^2	480	227	208	441
500^2	972	546	103	183
750^2	1477	819	68	122
1000^2	2055	1096	49	91
1250^2	2968	1378	34	73

Figure 4.5: Game of Life parallel row-per-thread method two: calculation time with a quadratic trend line and a linear trend line.



The second row-per-thread segmentation method utilizes all the SMs of a GPU for $N \geq S_c$, and divides the workload by dividing the number of rows in the grid equally between the number of SMs. Each SM must process $\frac{1}{S_c}N$ rows of N cells, for a total of $\frac{1}{S_c}N^2$ cells.

It is important to note how the value of N will influence the computation time. For $0 < N \leq 192S_c$, each SM will assign a single row of N cells to one of its cores. Assuming all cores process their work simultaneously, taking time t to process one cell, it will take time Nt to process all N^2 cells. For $N > 192S_c$ there are more rows

Stellenbosch University https://scholar.sun.ac.za

of cells than there are cores. For the first $192S_c$ number of rows, the calculation time will again be equal to Nt, assuming all work is processed simultaneously. The remaining $N - 192S_c$ rows are processed afterwards, which will steadily increase the computation time. The data from Table 4.7 reflect this trend as a near linear increase in the calculation time is noticed (also refer to Figure 4.5).

Computer A only has 384 CUDA cores, and thus from $N = 500$, an increase in the computation time is noticeable. Computer B, for which the computation time increases roughly at a constant rate, has 2304 CUDA cores and an increase in the computation time is only expected for $N > 2304$. In general, the time complexity function is $\frac{1}{192S_c}N^2t$, assuming it takes time t to process 192 cells. As with the time complexity function of the first row-per-thread method, a quadratic increase in the computation time is expected, as $N \to \infty$.

Note that, for all the experiments performed, all the SMs are utilized. The results show that the second row-per-thread segmentation method is faster than the first row-per-thread method. This occurs because more cells are processed simultaneously for smaller grid sizes. For Computer B, six times more cells will be processed simultaneously for larger grid sizes, where $N > 192S_c$.

Table 4.8 shows the data gathered for the cell-per-thread data segmentation method. Figure 4.6 shows the calculation time series.

Table 4.8: Results for Game of Life parallel cell-per-thread segmentation method experiments.

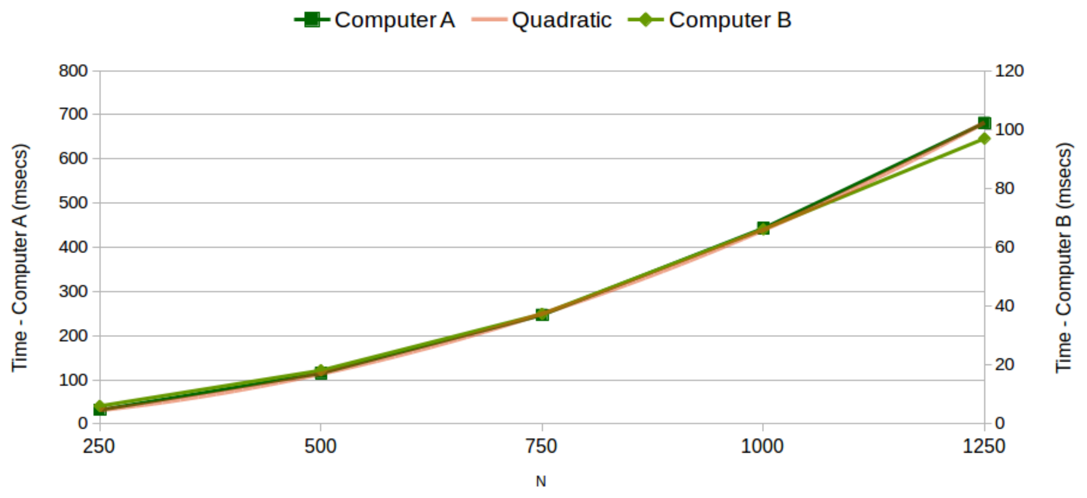
	Calculation time (msecs)		**States per second**	
Grid size	**Computer A**	**Computer B**	**Computer A**	**Computer B**
250^2	32	6.09	3160	16422
500^2	115	18	871	5497
750^2	247	37	405	2676
1000^2	443	66	226	1512
1250^2	682	97	147	1031

For the cell-per-thread segmentation method, the grid is divided into sub-grids of size $k \times k$. These sub-grids are then equally divided amongst the SMs, and each cell per sub-grid is assigned to a single thread. For an N^2 grid size, there are $\frac{1}{k^2}N^2$ sub-grids, each sub-grid containing k^2 cells. For S_c number of SMs, each SM has to process $\frac{1}{S_ck^2}N^2$ cells.

For the time complexity of the cell-per-thread segmentation method, the time taken for an SM to process its workload is measured first. Since a sub-grid is of size $k \times k$ where $k \leq 32$, a thread-block of the same size is assigned to each sub-grid. Thus each SM has to process k^2 cells (processing 192 cells simultaneously), which takes time t. Therefore it will take $\frac{1}{192}k^2t$ to process all cells per sub-grid. The

Stellenbosch University https://scholar.sun.ac.za

Figure 4.6: Game of Life parallel cell-per-thread method: calculation time with quadratic trend lines.



total time to process N^2 cells is $\frac{N^2}{S_c k^2} \times \frac{k^2}{192}t = \frac{1}{192 S_c}N^2 t$, which is the same as the time complexity function for the second row-per-thread segmentation method. The difference is that this time complexity is only achievable for large grid sizes for the row-per-thread methods, where more rows have to be processed. For the cell-per-thread method it is the average time complexity since all processing resources are utilized for all grid sizes.

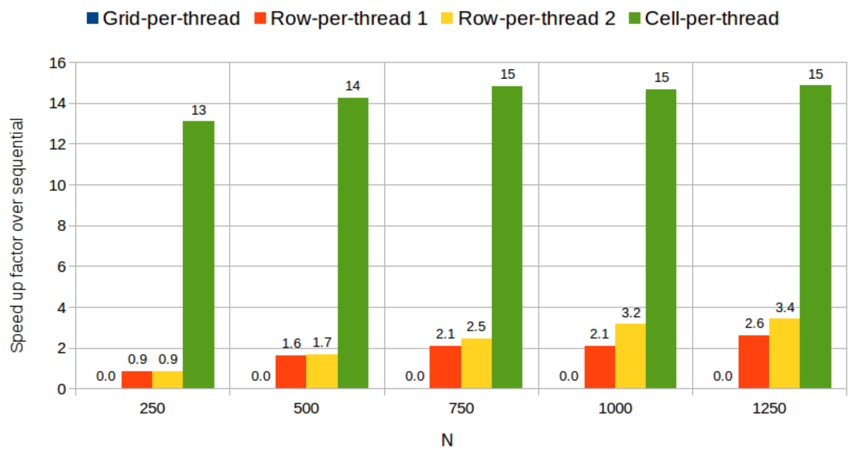
The dominating factor in the time complexity function is the grid size, N^2, whilst S_c stays constant for the GPU used. The computation time is expected to increase quadratically as the size of N is increased. Figure 4.6 shows this trend when comparing the calculation times with a quadratic trend line.

The data from Table 4.8 also shows results synonymous with quadratic growth. For Computer A, the growth in calculation time for $N = 250$ to $N = 500$ is 3.63, and for $N = 500$ to $N = 1000$, 3.85. For Computer B, the growth in calculation time for $N = 250$ to $N = 500$ is 2.99 and for $N = 500$ to $N = 1000$, 3.63. Both Computer A and Computer B have a calculation time growth less than the expected quadratic growth. The main factor for this result is the number of cells that can be processed simultaneously (Computer B having 6 times the number of cores that Computer A has). As N increases, the growth in computation time is expected to stabilize at a quadratic increase, since the number of processing cores stays constant.

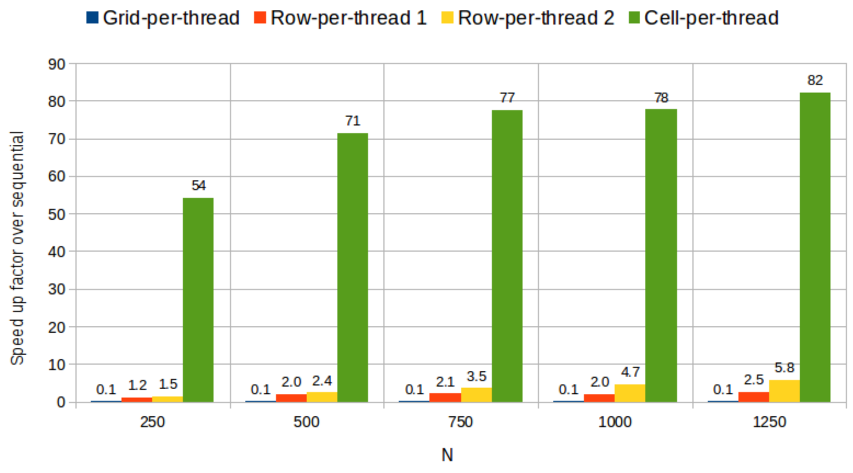
Stellenbosch University https://scholar.sun.ac.za

2.6 Sequential versus parallel experiment data

The experimental data shows that parallel implementations deliver better results when using more processing resources. However, when limiting the use of processing resources, the sequential implementation performs better. Figure 4.7 shows the speed up factor of each parallel data segmentation method over the sequential implementation.



(a) Computer A.



(b) Computer B.

Figure 4.7: Game of Life parallel segmentation methods: speed up factors over sequential implementation.

Stellenbosch University https://scholar.sun.ac.za

The grid-per-thread segmentation method does not produce any speed up over the sequential implementation since a minimal amount of GPU processing resources is used. For both Computer A and Computer B the sequential implementation outperforms the grid-per-thread method, where the sequential implementation is about 43 times faster than the grid-per-thread method for Computer A, and about 8.61 times faster for Computer B.

For both row-per-thread segmentation methods the GPU starts to outperform the sequential implementation. For the first row-per-thread method, the sequential implementation outperforms the parallel method for $N = 250$ for Computer A, and the parallel implementation only starts to deliver better performance for $N \geq 296$. For Computer B the parallel implementation is faster than the sequential implementation for $N = 250$, but is slower than the sequential implementation for $N \leq 210$. The sequential implementation is expected to outperform the parallel row-per-thread method for certain values of N, as there is an overhead when transferring the data to the GPU, and since the CPU has a higher clocks speed. The GPU needs to use more of its processing resources before it can deliver equivalent or better performance than the sequential implementation.

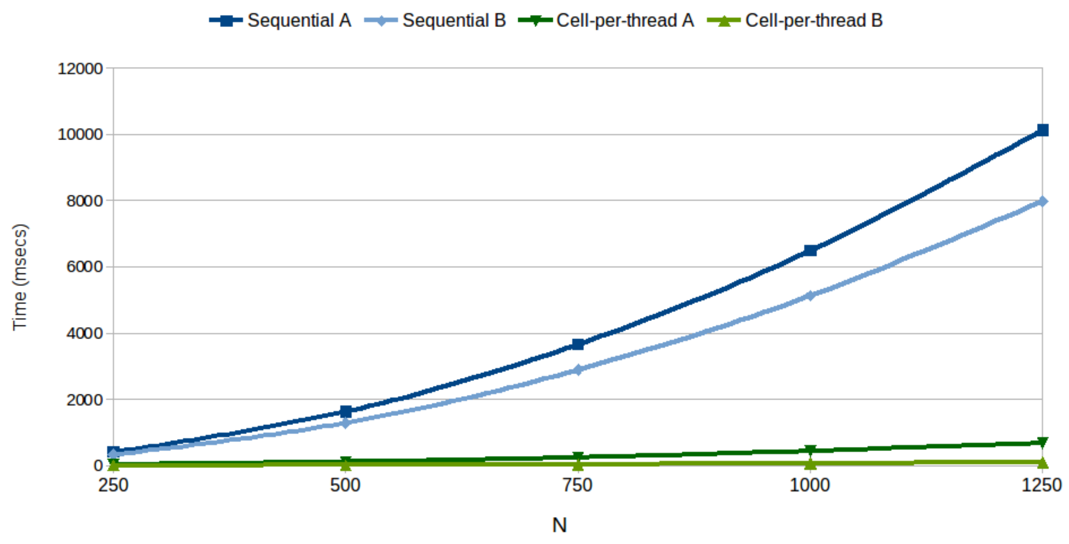
The same applies for the second row-per-thread segmentation method. For Computer A the sequential implementation outperforms the parallel implementation for $N = 250$, since the GPU utilizes less processing resources. The parallel implementation only starts to outperform the sequential implementation for $N \geq 296$. For Computer B, the sequential implementation outperforms the parallel implementation for $N \leq 135$, but is outperformed by the parallel implementation for $N > 135$.

For the cell-per-thread segmentation method, the parallel implementation starts to outperform the sequential implementation for $N \geq 13$ for Computer A and $N \geq 15$ for Computer B. For greater values of N the parallel implementation starts to outperform the sequential implementation by a significant margin. Figure 4.7 indicates a speed up factor of about 14 over the sequential implementation for Computer A, and between 50 and 80 for Computer B. The primary reason attributing to this observation is that the cell-per-thread method starts using all its available processing resources for any value of N, where applicable; for small values of N, where there are less cells than CUDA cores, all the cells are processed simultaneously.

It is also interesting to note that the cell-per-thread implementation performed on the less powerful GPU of Computer A, delivers better performance times than the sequential implementation performed on the more powerful CPU of Computer B. This is the case for all the experiments conducted. Figure 4.8 shows the performance times of the sequential implementations and the parallel cell-per-thread implementations for both Computer A and Computer B. Figure 4.8 shows a strong indication of the proficiency of the GPU when utilizing all available processing resources.

Stellenbosch University https://scholar.sun.ac.za

Figure 4.8: Game of Life sequential versus parallel cell-per-thread: calculation times for all experiments.



Stellenbosch University https://scholar.sun.ac.za

3 Clay deformation

The second CA that will be analyzed is a clay deformation simulation. The original idea behind this CA is based on three-dimensional free form shape modeling within a voxel space, and was proposed by Arata et al [3]. For this thesis a discrete two-dimensional model proposed by Druon et al will be used, which is based on the model proposed by Arata et al. In the clay deformation CA, an object is modeled by assigning 'clay' to certain cells during the initialization of the CA [10]. In order to deform the initial clay model, a 'plate' is used to push down on the clay model, which causes the cells directly impacted by this operation to have an overload of clay. Cells that are overloaded or overburdened with clay, shift portions of clay to neighbouring cells.

3.1 Rules

For the clay deformation CA, each cell can have one three states:

- 0 – the cell is empty (not filled with clay);
- 1 – the cell contains a specific amount of clay; or
- 2 – the cell is part of a plate and contains no clay.

Cells containing clay can either be stable or overburdened, which means that the amount of clay contained in the cell is either below or above a certain threshold, respectively. If the clay-containing cells in the CA are stable, the clay model will not be subjected to deformation. If, however, some cells become overburdened, these cells will distribute a specified amount of clay to its neighbours that are not overburdened. The process of distributing clay from an overburdened cell to its stable neighbouring cells is performed when executing the repartition rule function of the CA.

The type of neighbourhood used for clay deformation is a Margolus neighbourhood [3], presented in Figure 4.9. The Margolus neighbourhood was created to accommodate partitioning CA. A partitioning CA is based on the following three attributes:

- the grid of cells is partitioned into disjoint cell blocks;
- each cell block is processed separately with a block rule (identical to a rule function of the CA); and
- the partitioning changes for each subsequent step of rule application, in order to have overlap between blocks [50].

Stellenbosch University https://scholar.sun.ac.za

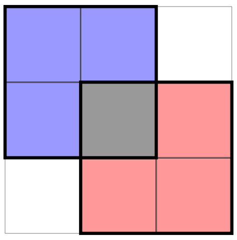


Figure 4.9: Two-dimensional Margolus neighbourhood: blue cells are processed during odd steps and red cells are processed during even steps, along with the gray (pillar) cell during execution of the repartition function.

A two-dimensional Margolus neighbourhood groups cells into blocks of size 2×2. Blocks of size 2×1, 1×2, or 1×1 are used for border cases where cells cannot be divided into 2×2 cell blocks. For clay deformation, the block rule is applied during the repartitioning function, for a number of steps. For even steps, a cell block is created by expanding a cell block from a pillar cell to the bottom-right diagonal cell of the pillar cell (refer to the red cell block in Figure 4.9). Figure 4.10 shows the cell blocks of an 8×3 grid, for an even step.

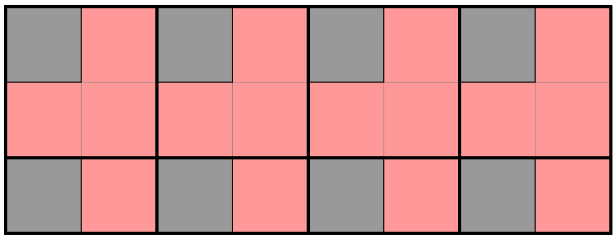


Figure 4.10: 8×3 grid of cells with 8 cell blocks for an even step, where all pillar cells are marked as gray.

For odd steps, a cell block is created by expanding a cell block from a pillar cell to the top-left diagonal cell of the pillar cell (refer to the blue cell block in Figure 4.9). Figure 4.11 shows the cell blocks of an 8×3 grid, for an odd step. The pillar cells in the last column are created to compensate for border cases, where none of the original pillar cells include the cells from the last column in its cell blocks. By creating these additional pillar cells, all cells are now part of cell blocks and can be processed.

When initializing the grid of cells, all cells containing clay have a stable amount of clay. In order to cause overburdened cells, a plate is used to push down on the top row of cells that contain clay. A plate is an $x \times 1$ row of cells, all with a state of 2 and that do not contain clay. When the plate pushes down on the top row of clay-filled cells, the clay of each cell is transferred to the cell below it

Stellenbosch University https://scholar.sun.ac.za

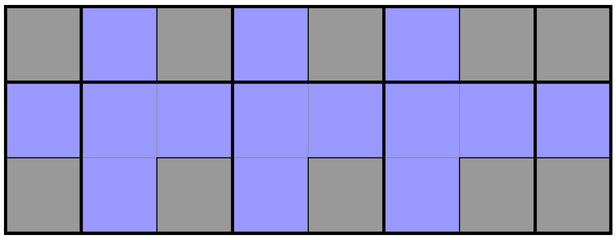


Figure 4.11: 8×3 grid of cells with 8 cell blocks for an odd step, where all pillar cells are marked as gray.

thereby creating overburdened clay cells. After the plate push operation is finished, the repartitioning operation is started. During this operation the repartitioning function is executed on the entire grid to distribute clay from overburdened cells to neighbouring cells. The repartitioning operation runs for i iterations (steps) or until the clay model is in a stable state before the i'th iteration is reached. Each iteration marks an even or an odd step. This entire process—the plate push operation and performing the repartitioning operation— is performed to calculate one generation of the clay deformation CA.

3.2 State calculation analysis

For clay deformation, most state calculations will be performed on cell blocks of size 2×2, as discussed in Section 3.1. Before a cell block can be processed, the size of the cell block must first be determined. A cell block is the even or odd Margolus neighbourhood of a pillar cell. For each pillar cell, a conditional check is performed to determine if the current repartitioning step is even or odd. Following this check, two conditional checks are performed to determine if a pillar cell can expand its Margolus neighbourhood one cell horizontally and one cell vertically (whether or not the pillar cell is on the border of the grid). To expand the neighbourhood in either direction, requires incrementing an integer assigned to each direction.

Once the dimensions of a cell block have been determined, the coordinates of the cell block are sent to the rule application function, which processes the cells in the cell block. The rule application function is divided into two parts. For the first part, a check is performed to determine if the cell block only consists of a pillar cell. If this is the case, an additional check is performed to determine of the pillar cell is overburdened. If the cell block consists of more than one cell, the number of cells in the cell block is determined by using simple integer arithmetic. These operation are all performed per pillar cell. The following operations are performed per cell. Each overburdened cell in the cell block is determined, by checking the amount of clay (stored as a `float`) against the threshold value, with a conditional check. The amount of clay of overburdened cells is multiplied by a constant `float` value, and the result is added to another `float` which stores the aggregate value

Stellenbosch University https://scholar.sun.ac.za

of all overburdened cells. This aggregate value is the amount of clay that will be distributed to stable clay cells. Finally, one of three integers values are adjusted depending on the state of the cell and the amount of clay.

For the second part of the rule application function a check is first performed to determine if all cells are overburdened. A similar check is performed to determine if the number of cells below the threshold is equal to the number of cells in the cell block. In either case, no operations can be performed on the cell block, in terms of clay redistribution, and a function return is performed. For cell blocks with both overburdened cells and stable cells, the clay redistribution process is performed. First, the amount of clay that must be distributed is divided by the number of stable cells, which requires a floating point operation. For all stable cells, the amount of clay that each cell must receive is added to the current amount of clay of each cell, which is one floating point operations. For overburdened cells, an appropriate amount of clay to subtract is calculated and then subtracted from the current amount of clay of each overburdened cell, which requires two floating point operations. In addition, a check is performed to determine if any of the cells are overburdened after the process of clay redistribution. If any cell is overburdened, a boolean value is updated.

The first part of the rule function will always be executed. If an $N \times N$ grid is perfectly divisible by 2×2 cell blocks, then for each pillar cell, five integer operations and three conditional checks must be performed. For each cell in the cell block, one conditional check, two floating point operations (if a cell is overburdened), and one integer operation must be performed.

The second part of the rule function will only be performed conditionally, which is determined by two conditional checks per pillar cell. Before clay redistribution, one floating point operation is performed per pillar cell. During the clay redistribution process two floating point operations are performed for overburdened cells, and one floating point operation is performed for stable cells. In addition, for each cell two conditional checks are performed.

In the worst case scenario, where both parts of the algorithm are executed, six integer operations, nine conditional checks, and ten floating point operations must be performed for each of the $\frac{1}{4}N^2$ cell blocks. All operations are performed in constant time. Compared to the Game of Life CA, less operations are performed per cell for the clay deformation CA. However, the additional operations for each pillar cell are also performed. For an $N \times N$ clay deformation CA, the rule application function is applied to $\frac{1}{4}N^2$ cell blocks. The dominating factor is N^2 and therefore the time complexity for all experiments is expected to be $O(N^2)$.

3.3 Experimental setup

All the clay deformation CA experiments are performed for the grid sizes 250^2, 500^2, 750^2, 1000^2 and 1250^2 to test the scalability and to investigate the time

Stellenbosch University https://scholar.sun.ac.za

complexity of of each implementation. For each grid size, ten generations of the CA is simulated. The total time to calculate ten generations is measured in milliseconds. This process is repeated ten times to validate the calculation times measured.

3.4 Sequential implementation

In this subsection the sequential implementation of the clay deformation CA is considered.

3.4.1 Sequential code extracts

The sequential algorithm is encompassed in the `gridNextStateSEQ()` function of the derived clay deformation `Clay` class in Listing B.7. This function iterates over every pillar cell, determines the Margolus neighbourhood of the pillar cell (the cell block coordinates), and sends the information to the rule application function, also known as the clay redistribution function. Pillar cells are fixed cells in the grid, located at each intersection of the even numbered rows and even numbered columns on the CA grid.

The clay redistribution function is given in Listing B.8. An overview of this function is given in Section 3.2. This function essentially determines whether there are any cells that are overburdened and if there are any cells to which the excess amount of clay can be transferred. A check must also be performed to determine if any of the cells in the cell block are part of a plate (has a state of '2'), since no clay can be transferred to a cell which is part of a plate.

For a scenario where clay can be transferred from overburdened cells to stable cells, the amount of clay of each cell in the cell block is updated. The sequential algorithm terminates once the clay model is stable or if i iterations of the clay redistribution function has been applied to clay model.

3.4.2 Sequential experiment results

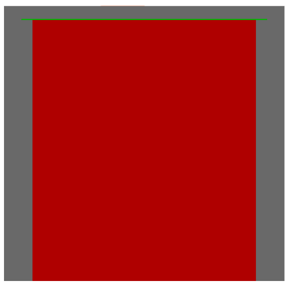
It is important to note that the number of steps of clay redistribution will have a noticeable impact on the time taken to calculate a single generation of a clay deformation CA. If no limit is used on the number of steps for the clay redistribution process, the time needed to calculate even a single generation could take exceedingly long. To demonstrate this, a clay deformation CA with a grid size of 250^2 is used to simulate 100 generations. The clay model is in the shape of a column and spans 80% the width of the grid and 95% the height of the grid. This experiment is performed on Computer B and is repeated for the following number of redistribution step limits: 1, 50, 100, 150, 200, 250, and 300. The results of this experiment are given in Table 4.9.

Stellenbosch University https://scholar.sun.ac.za

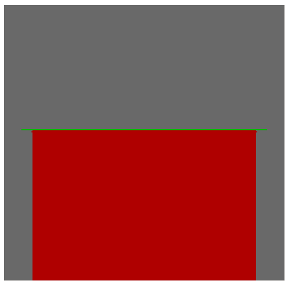
Table 4.9: Sequential redistribution step experiment: average calculation times for different step limits in milliseconds.

Step limit	Calculation time
1	58
50	3807
100	8361
150	12735
200	16847
250	20626
300	24252
No limit	411716

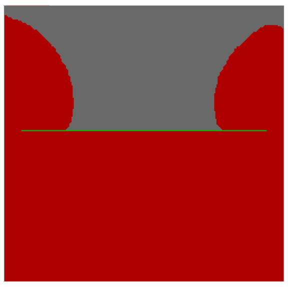
Figure 4.12: Clay redistribution for one step limit and for no limit, for 100 generations.



(a) Original model.



(b) One step deformed model.



(c) Completely deformed model.

When only one clay redistribution step is performed, the total time needed to calculate 100 generations is faster than when calculating 100 generations for a Game of Life CA with the same grid size. However, very little clay is distributed and no real deformation is observed. If no limit is placed on the number of redistribution steps, a completely deformed clay model is produced (refer to Figure 4.12). However, it takes approximately 6 minutes and 52 seconds to calculate 100 generation when using no limit. By increasing the number of steps by a constant number, the

Stellenbosch University https://scholar.sun.ac.za

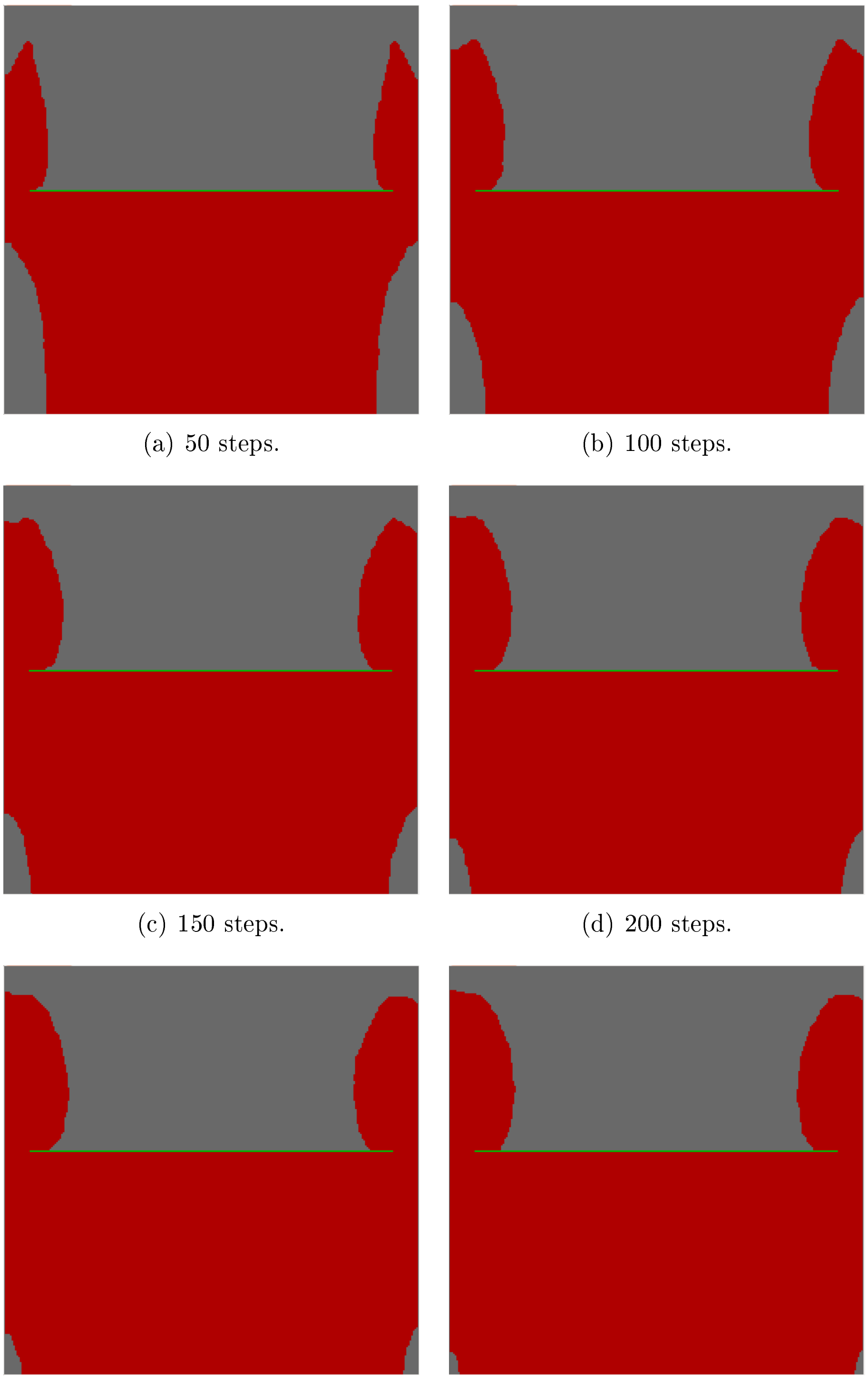

(a) 50 steps.
(b) 100 steps.
(c) 150 steps.
(d) 200 steps.
(e) 250 steps.
(f) 300 steps.

Figure 4.13: Clay deformation for different redistributed step limits.

Stellenbosch University https://scholar.sun.ac.za

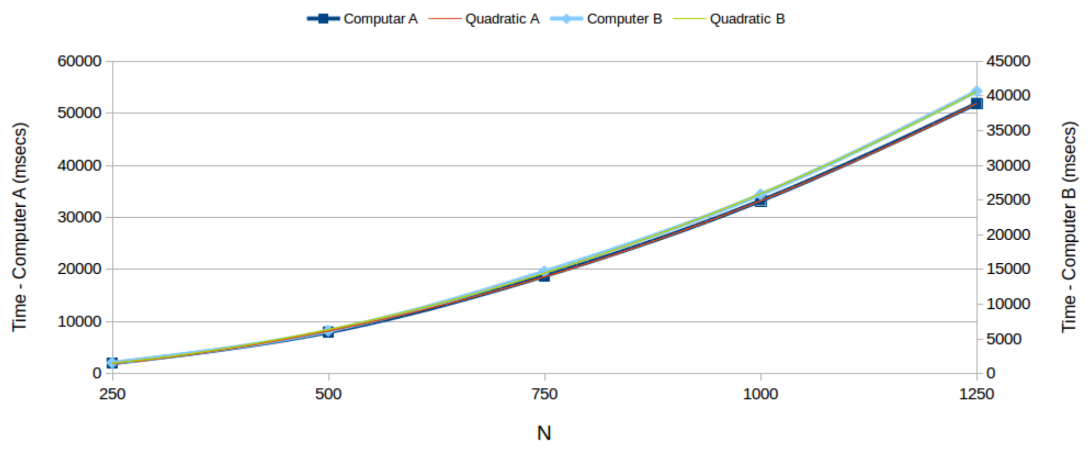
calculation time increases by a near constant amount, and the clay model becomes more deformed. Figure 4.13 shows the deformation of the clay model for each redistribution step limit, for 100 generations. By 300 steps of redistribution, the clay model is nearly as deformed as when no redistribution limit is applied.

For all the clay deformation CA experiments performed, a 300 step redistribution limit is used. Table 4.10 shows the average calculation times for each grid size and the average number of CA generations calculated per second, for both Computer A and Computer B. Figure 4.14 shows the calculation time series. First

Table 4.10: Results for clay deformation sequential experiments.

	Calculation time (msecs)		Generations per second	
Grid size	**Computer A**	**Computer B**	**Computer A**	**Computer B**
250^2	2033	1585	4.92	6.31
500^2	7996	6188	1.25	1.62
750^2	18774	14728	0.53	0.68
1000^2	33215	25803	0.30	0.39
1250^2	51852	40654	0.19	0.25

Figure 4.14: Clay deformation sequential calculation time with quadratic trend lines.



notice that the time needed to calculate ten generations of clay deformation is considerably more than the time needed to calculate 100 generations for the Game of Life CA. This is because a maximum of 300 steps for the redistribution of clay is

Stellenbosch University https://scholar.sun.ac.za

used. If it takes time t to process one cell block during one step of clay redistribution, it will take time $\frac{1}{4}N^2t$ to process all cell blocks. This process will be repeated a maximum of 300 times for each generation of clay deformation. Thus the total time to calculate one generation is equal to $75N^2t$. The dominating factor in this formula is N^2, and thus a quadratic time increase is expected as the value of N increases.

When referring to the results in Table 4.10, notice that the time increase constant for $N = 250$ to $N = 500$ is 3.93 and 3.90 for Computer A and Computer B, respectively. The time increase constant for $N = 500$ to $N = 1000$ is 4.15 and 4.16 for Computer A and Computer B, respectively. In both cases the value of N is doubled and a quadratic time increase is expected, which is reflected in the results.

3.5 Parallel implementation

In this subsection the parallel implementations for each of the data segmentation methods used for the clay deformation CA is considered.

3.5.1 Parallel code extracts

The four data segmentation methods for the parallel algorithms largely stay the same as for the Game of Life CA experiments. There is one difference that comes into play for the row-per-thread and cell-per-thread methods, which has to do with the fact that threads now process cell blocks instead of cells. For the row-per-thread methods, the work is divided into rows of cell blocks, and each row is assigned to a thread. For the cell-per-thread method, a cell block is assigned to each thread.

Overall, the grid-per-thread method is similar to the method used for Game of Life. The only difference is the change in the for-loop iterations to compensate for the fact that pillar cells are being worked with, and that the cell blocks are created from each pillar cell. The grid-per-thread segmentation method is provided in Listing B.9.

The row-per-thread data segmentation methods for clay deformation use the same parallel algorithm, which is included in Listing B.10. The difference between the methods is based on how many threads and thread-blocks are utilized during the function call, defined in the angle brackets. The first row-per-thread segmentation method, which utilizes 1024 threads per thread-block, assigns one thread to one row of cell blocks. One SM will thus process a maximum of 2048 rows of cells. The second row-per-thread segmentation method divides the number of rows equally among the SMs.

The cell-per-thread data segmentation method for clay deformation assigns one thread to one pillar cell or cell block of 2×2 cells. As discussed in Section 2.5.1, thread-blocks of size 16×16 are used. Each thread-block will process 4×16^2 cells. The parallel algorithm used for the clay deformation cell-per-thread method

Stellenbosch University https://scholar.sun.ac.za

is included in Listing B.11.

The rule application function referred to in each of the segmentation methods is identical to the rule application function used for the sequential implementation (refer to Listing B.8).

3.5.2 Parallel experiment results

For the grid-per-thread method the experiments were only performed once per grid size. This decision was made because the time needed to calculate ten generations for any grid size was far greater than the influence of background processes running on the systems could have on the total calculation time. Table 4.11 shows the data gathered for the grid-per-thread data segmentation method. Figure 4.15 shows the calculation time series. First notice the calculation time needed to calculate ten

Table 4.11: Results for clay deformation parallel grid-per-thread segmentation method experiments.

	Calculation time (msecs)		Generations per second	
Grid size	**Computer A**	**Computer B**	**Computer A**	**Computer B**
250^2	301060	34774	0.0330	2.88
500^2	1220276	135301	0.0082	0.74
750^2	2857308	309091	0.0035	0.32
1000^2	5045249	537198	0.0020	0.19
1250^2	7890809	850924	0.0013	0.12

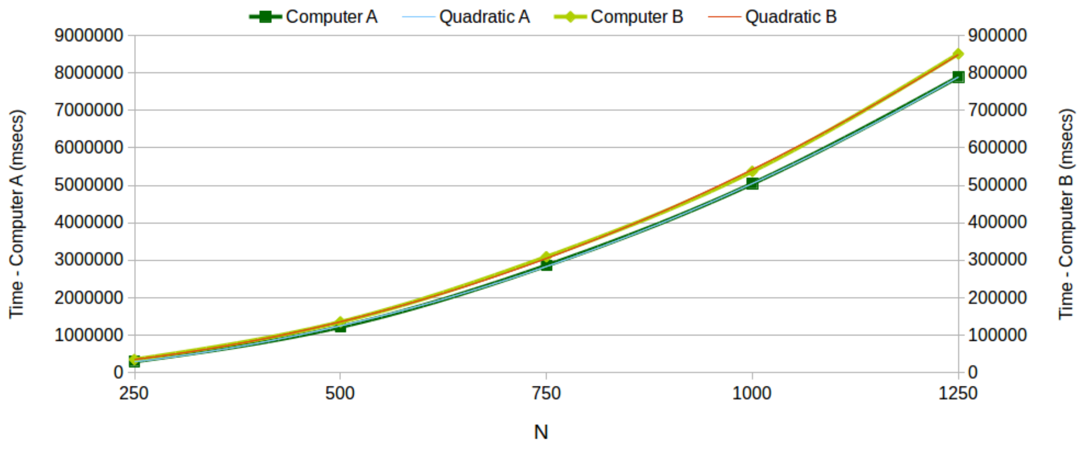
generations, and the average number of generations that are processed per second. For the smallest grid size of 250^2, the inefficiency of this segmentation method is significant, especially for Computer A[1].

When analyzing the time increase between $N = 250$ to $N = 500$, a time increases constant of 4.05 and 3.89 is observed for Computer A and Computer B, respectively. The time increase constant between $N = 500$ to $N = 1000$ is 4.13 and 3.97 for Computer A and Computer B, respectively. These time increase constants suggest a quadratic time complexity function. Since the data is divided equally between the number of SMs (S_c), each SM has to process $\frac{1}{4S_c}N^2$ cell blocks during one clay redistribution step. If it takes time t to process one cell block, and since the redistribution process is repeated for a maximum of 300 steps, it will take time equal to $\frac{1}{S_c}75N^2t$ to calculate one generation per SM. The dominating factor in this formula is N^2, which confirms a quadratic time increase when analyzing the data. Note the close correlation between the time increase and a quadratic function

[1]Performance speed analysis between the sequential and parallel algorithms are given in Section 3.6

Stellenbosch University https://scholar.sun.ac.za

Figure 4.15: Clay deformation parallel grid-per-thread method: calculation time with quadratic trend lines.



as is shown in Figure 4.15.

Table 4.12 shows the data gathered for the first row-per-thread data segmentation method. Figure 4.16 shows the calculation time series.

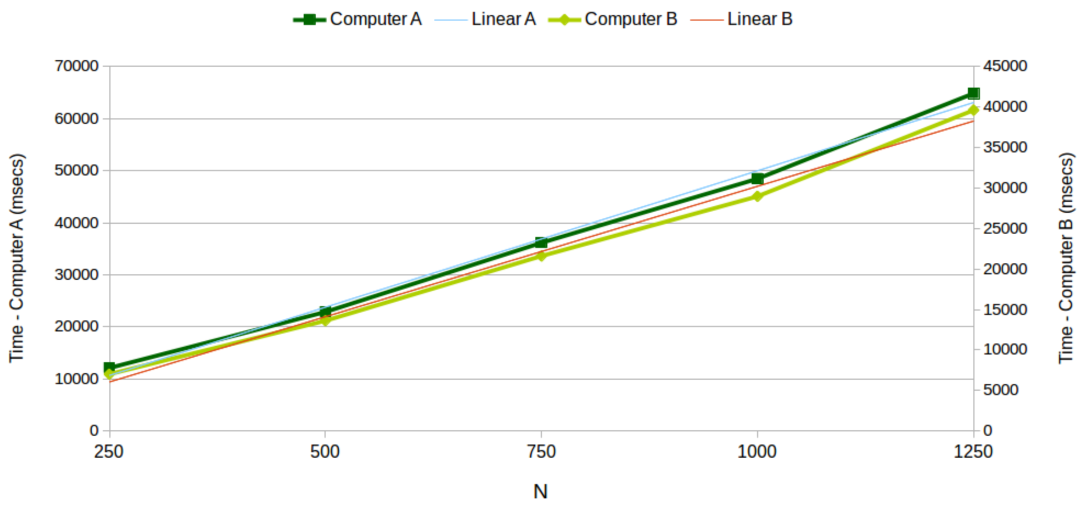
Table 4.12: Results for clay deformation parallel row-per-thread segmentation method one experiments.

	Calculation time (msecs)		Generations per second	
Grid size	**Computer A**	**Computer B**	**Computer A**	**Computer B**
250^2	12090	7007	0.83	1.43
500^2	22855	13610	0.44	0.73
750^2	36154	21600	0.28	0.46
1000^2	48449	28972	0.21	0.35
1250^2	64834	39609	0.15	0.25

For the first row-per-thread method, a maximum of 2048 rows of cells are assigned to one SM, since a single thread covers two rows of cells and because the thread limit per thread-block is 1024. Each thread effectively processes a single row of cell blocks. All experiments are therefore performed with one SM. For $N \leq 2048$, one SM processes all $\frac{1}{4}N^2$ cell blocks; each thread processing $\frac{1}{2}N$ cell blocks, or $2N$ cells. For $N > 2048$ an additional SM is used to process the remaining rows, and the first SM will process $512N$ cell blocks. In general, an SM will process $\frac{1}{4S_c}N^2$ cell blocks.

Stellenbosch University https://scholar.sun.ac.za

Figure 4.16: Clay deformation parallel row-per-thread method one: calculation time with linear trend lines.



Note that a total of 192 cell blocks are processed simultaneously, since an SM has 192 cores. If it takes time t to process 192 cell blocks per redistribution step, it will take time $\frac{300}{768}N^2t$ for one SM to process one generation, for a maximum of 300 redistribution steps, where $N \leq 2048$. In general, it will take time $\frac{300}{768S_c}N^2t$ to process one generation, where $N > 2048S_c$. The dominating factor in this formula is N^2, and thus the time complexity for the row-per-thread method is $O(N^2)$. A quadratic increase in the calculation time is expected as the value of N is increased.

However, when analyzing Table 4.12 the data yields a time increase constant of 1.89 and 1.94 between $N = 250$ to $N = 500$, for Computer A and Computer B, respectively. Between $N = 500$ to $N = 1000$ the time increase constant is 2.12 and 2.13 for Computer A and Computer B, respectively. These results suggest a more linear increase in the calculation time as the value of N increases (see Figure 4.16).

This is because a number of rows of cell blocks are processed simultaneously, where each row has $\frac{1}{2}N$ cell blocks. If N increases by a constant number, the number of cell blocks per row also increases by a constant number. However, more rows of cell blocks must also be processed, thus the calculation time gradually increases. Notice that between $N = 250$ to $N = 500$ and $N = 500$ to $N = 1000$ there is in increase (albeit slightly) in the calculation times as N is doubled. Thus, as $N \to \infty$, the calculation time is expected to stabilize at a quadratic time increase, since the GPU processing resources stay constant.

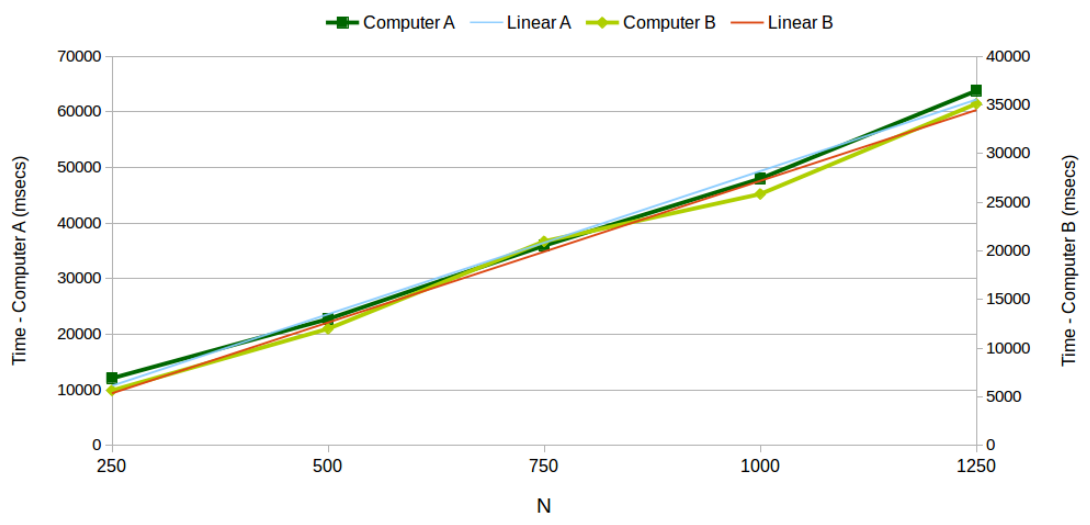
Table 4.13 shows the data gathered for the second row-per-thread data segmentation method. Figure 4.17 shows the calculation time series. The second row-per-

Stellenbosch University https://scholar.sun.ac.za

Table 4.13: Results for clay deformation parallel row-per-thread segmentation method two experiments.

	Calculation time (msecs)		Generations per second	
Grid size	Computer A	Computer B	Computer A	Computer B
250^2	12084	5652	0.83	1.77
500^2	22736	11994	0.44	0.83
750^2	35950	20979	0.28	0.48
1000^2	47991	25841	0.21	0.39
1250^2	63805	35118	0.16	0.28

Figure 4.17: Clay deformation parallel row-per-thread method two: calculation time with linear trend lines.



thread method divides the rows of the grid equally between the number of SMs. Each SM has to process $\frac{1}{2S_c}N$ rows of cell blocks with $\frac{1}{2}N$ cell blocks per row, for a total of $\frac{1}{4S_c}N^2$ cell blocks.

All SMs are used for all values of N (but not necessarily all processing cores), thus more rows of cell blocks will be processed simultaneously than for the first row-per-thread method. Since each SM can process 192 rows of cell blocks simultaneously (or 384 rows of cells), only one row of cell blocks will be assigned to a single processing core for $0 < N \leq 384S_c$. When $N \leq 384S_c$, if it takes time t to process one cell block for one step of clay redistribution, it will take time $150Nt$ to process one generation with a 300 redistribution step limit. For $N > 384S_c$, each processing core that has finished processing its row of cell blocks, is assigned to an unprocessed row of cell blocks until all the rows have been processed. As

Stellenbosch University https://scholar.sun.ac.za

N increases, each processing core has to process an additional constant number of cell blocks per row (while more processing cores will be utilized), thus steadily increasing the load for each processing core.

Figure 4.17 tends to show a linear increase in the calculation time, as can expected for the grid sizes used. When analyzing the data, the time increase constants between $N = 250$ to $N = 500$ are 1.88 and 2.12 for Computer A and Computer B, respectively. Between $N = 500$ to $N = 1000$ the time increase constants are 2.11 and 2.15 for Computer A and Computer B, respectively. These results coincide with the linear time increase. However, as with the first row-per-thread segmentation method, the time increase constants do increase as N is doubled. And thus, for $N >> 384S_c$, the calculation time is expected to stabilize at a quadratic time increase, since the GPU processing resources stay constant.

Table 4.14 shows the data gathered for the cell-per-thread data segmentation method. Figure 4.18 shows the calculation time series.

Table 4.14: Results for clay deformation parallel cell-per-thread segmentation method experiments.

	Calculation time (msecs)		**Generations per second**	
Grid size	**Computer A**	**Computer B**	**Computer A**	**Computer B**
250^2	520	137	19	73
500^2	1767	282	5.66	35
750^2	3780	561	2.64	18
1000^2	6556	942	1.53	11
1250^2	10056	1437	0.99	6.95

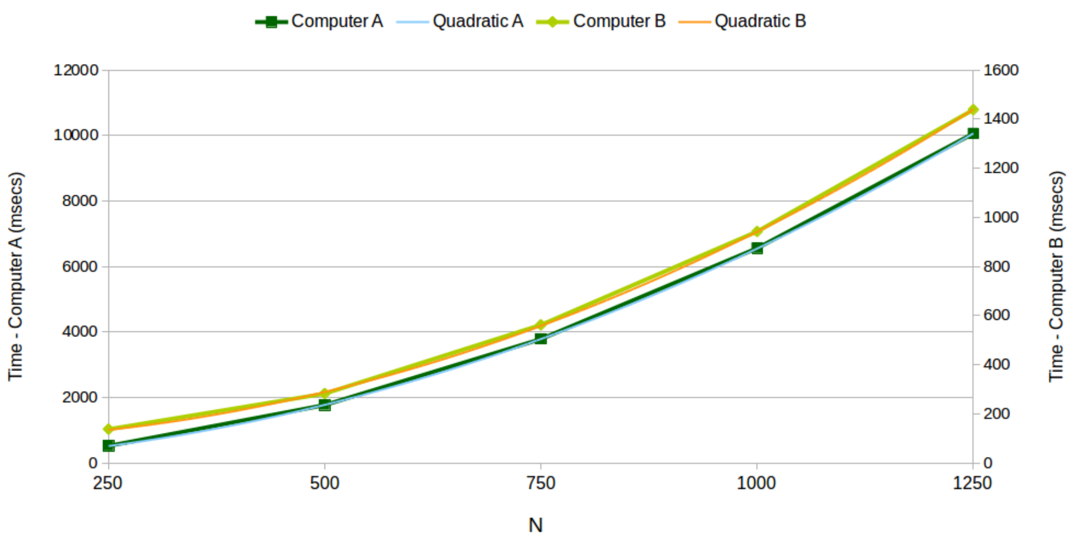
For clay deformation the cell-per-thread segmentation method uses thread-blocks of size $k \times k$ to process the $\frac{1}{4}N^2$ cell blocks. Each thread per thread-block is assigned to one cell block, thus $\frac{1}{4k^2}N^2$ thread-blocks is needed to process all N^2 cells. For S_c SMs, each SM will process $\frac{1}{4k^2S_c}N^2$ thread-blocks.

When analyzing the time complexity of the cell-per-thread method, one must first analyze the time it takes for an SM to process one thread-block for one redistribution step. An SM can process 192 threads simultaneously, taking time t. To process an entire thread-block will thus take time $\frac{1}{192}k^2t$, or $\frac{300}{192}k^2t$ for a 300 redistribution step limit. In general, since all the processing resources are used, the total time to calculate one generation is equal to $\frac{300}{768S_c}N^2t$. As with the other segmentation methods, a quadratic time increase is expected as the value of N is increased.

Figure 4.18 shows the close correlation between the calculation time series of both Computer A and Computer B, and a quadratic trend line. This figure also shows that the cell-per-thread segmentation method performs according to its time

Stellenbosch University https://scholar.sun.ac.za

Figure 4.18: Clay deformation parallel cell-per-thread method: calculation time with quadratic trend lines.



complexity function. The results of Table 4.14 indicate time increases constants of 3.40 and 2.06 between $N = 250$ to $N = 500$ for Computer A and Computer B, respectively. The time increase constants between $N = 500$ to $N = 1000$ are 3.71 and 3.34 for Computer A and Computer B, respectively. Since the GPU of Computer B has more SMs than the GPU of Computer A, the workload per SM is less. Thus, a slower increase in the convergence of the calculation time for Computer B, with the expected quadratic time-increase is observed. For Computer A (which has six times less SMs than Computer B) the time-increase constant more quickly converges with the theoretical quadratic time increase.

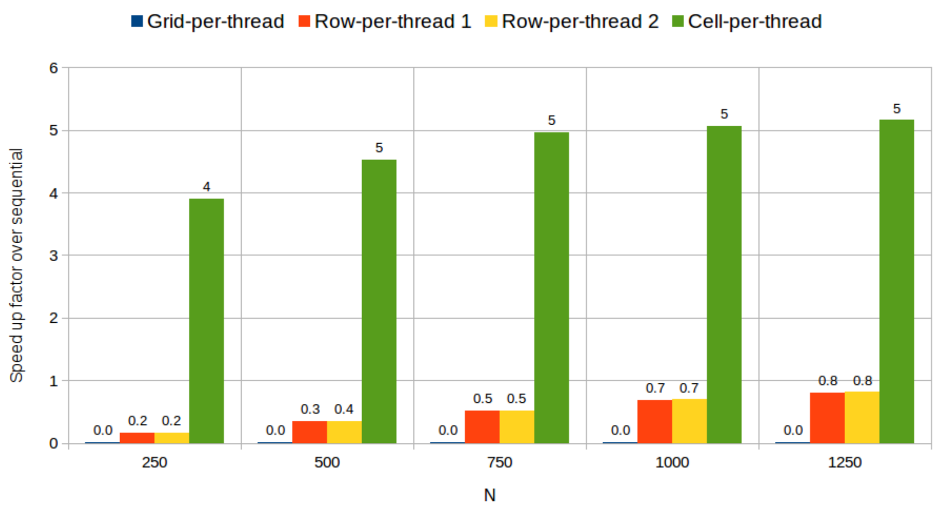
3.6 Sequential versus parallel experiment data

Figure 4.19 shows the speed up factor of each parallel data segmentation method over the sequential implementation.

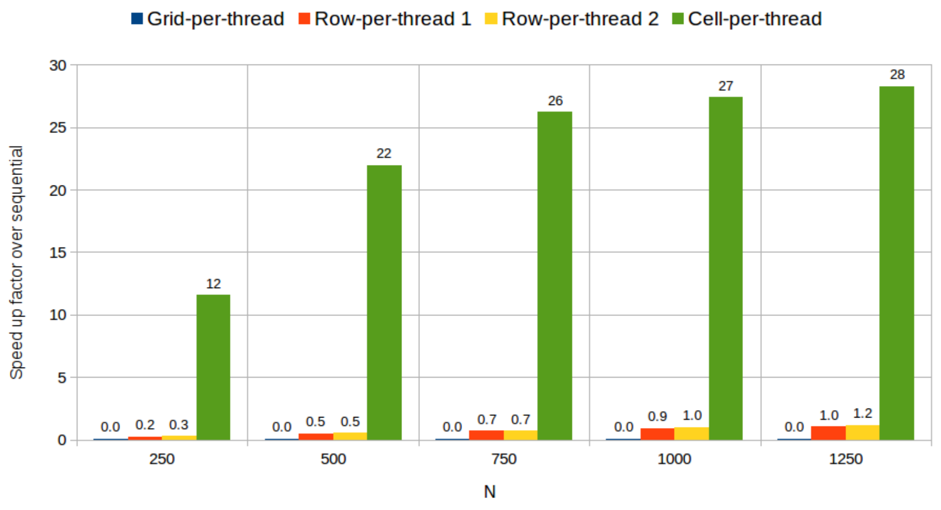
As with the Game of Life CA experiments, the grid-per-thread method is the most inefficient method when processing a CA. Limiting the use of processing resources on the GPU to only one thread per SM, severely slows the calculation time. Figure 4.19 indicates speed up factors all below 1, for the grid-per-thread method. The sequential implementation is about 152 times faster than grid-per-thread method for Computer A, and about 21 times faster for Computer B.

In contrast to speed up provided by the row-per-thread implementations for the Game of Life CA, the more computationally involved rule application function of

Stellenbosch University https://scholar.sun.ac.za



(a) Computer A.

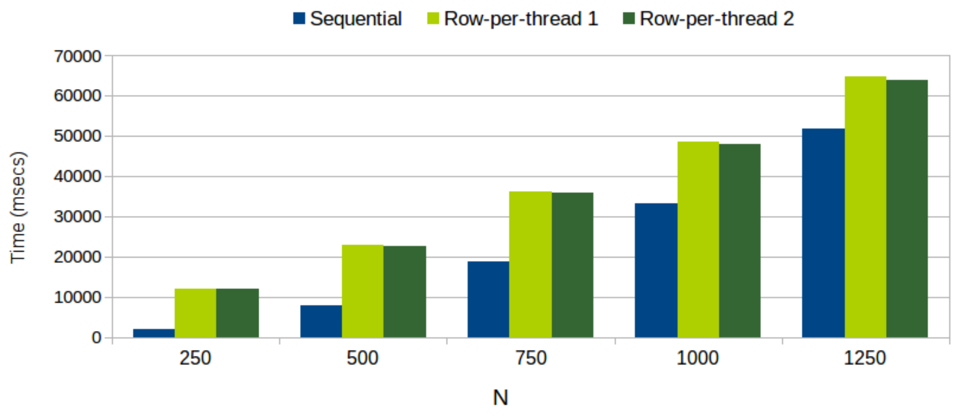


(b) Computer B.

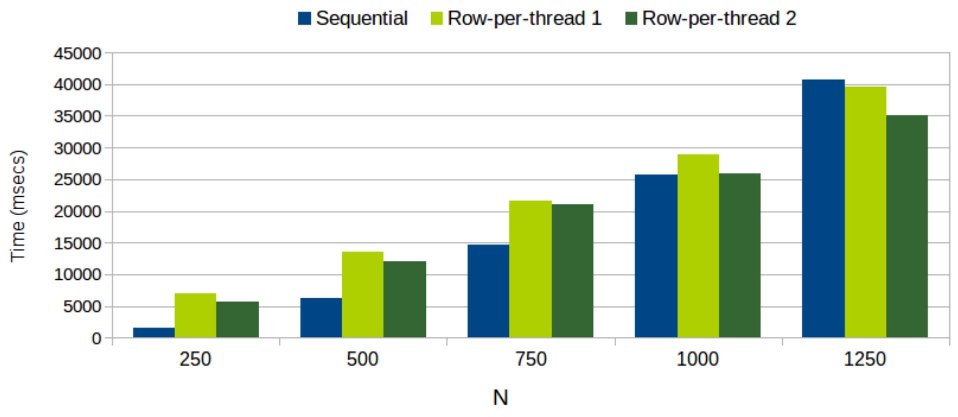
Figure 4.19: Clay deformation parallel segmentation methods: speed up factors over sequential implementation.

the clay deformation CA, hampers the performance of the row-per-thread implementations. When increasing the use of processing resources as is done with the first row-per-thread method, the sequential implementation still outperforms the particular parallel implementation for Computer A. When comparing the quotient between the calculation times of parallel and sequential implementations, the cal-

Stellenbosch University https://scholar.sun.ac.za



(a) Computer A.



(b) Computer B.

Figure 4.20: Clay deformation sequential and parallel row-per-thread methods: calculation times plotted for different values of N.

culation time of the parallel implementation starts to converge with the calculation time of the sequential implementation, for $N > 1250$. Thus, the row-per-thread implementation is expected to outperforming the sequential implementation for larger grid sizes. For Computer B, the first row-per-thread parallel implementation is slower than the sequential implementation, and only starts to outperform the sequential implementation for $N \geq 1250$.

For the second row-per-thread method, the sequential implementation also outperforms the parallel implementation in all the experiments performed on Computer A. The quotient between the calculation times of the parallel and sequential implementations, also decreases for larger grid sizes, and the calculation time of the

Stellenbosch University https://scholar.sun.ac.za

parallel implementation is expected to converge with the calculation time of the sequential implementation, for $N > 1250$. For Computer B, the sequential implementation outperforms the parallel implementation for $N < 1020$. For $N \geq 1020$, the parallel implementation starts to outperform the sequential implementation, and is about 1.16 times faster than the sequential implementation, for $N = 1250$. Figure 4.20 compares the calculation times of the sequential implementation and the parallel row-per-thread methods.

Finally, the cell-per-thread implementation, which utilizes all of the GPU processing resources, is faster than the sequential implementation for all values of N. Figure 4.19 indicates a speed up factor between 4 and 5 over the sequential implementation for Computer A, and between 10 and 25 for Computer B. For very small grid sizes, where the overload of copying data to the GPU is more significant than for larger grid sizes, the sequential implementation outperforms the parallel implementation for $N \leq 76$ for Computer A, and $N \leq 64$ for Computer B. Finally, as with the Game of Life CA experiments, the cell-per-thread implementation performed on the less powerful GPU of Computer A, delivers better performance times than the sequential implementation performed on the more powerful CPU of Computer B. For $N = 1250$, the GPU of Computer A outperforms the CPU of Computer B by about 30 seconds (refer to Figure 4.21).

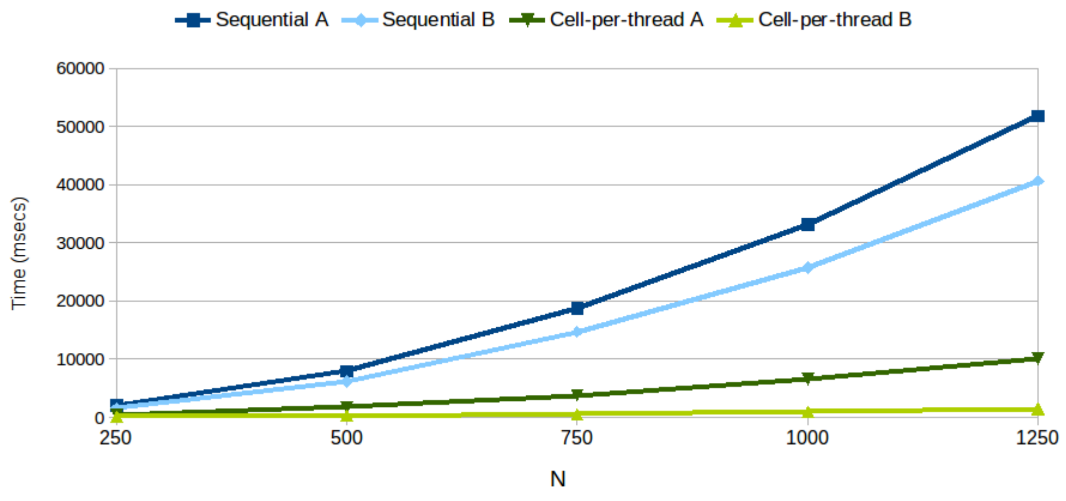


Figure 4.21: Clay deformation sequential and parallel cell-per-thread method: calculation times plotted for different values of N.

Stellenbosch University https://scholar.sun.ac.za

4 Ant clustering

The third CA that will be analyzed is based on sorting a complex data set, by clustering similar data points [31, 41]. The algorithm proposed by Lumer et al, known as the LF algorithm, refers to data points as objects that are scattered on a two-dimensional grid. A number of ants are then placed on the grid at random locations, and are used to relocate the objects. The ultimate objective is for the ants to create the least number of clusters of similar objects, which implies better clustering. Once an ant encounters an object, it might pick up the object depending on the density of similar objects in the neighbourhood of the object in question. The ant will be more likely to pick up the object if there are not many similar objects in its neighbourhood. If the ant has picked up the object, the ant will move around on the grid and drop the object based on a similar principle. If the ant has located to an area on the grid that contains many objects similar to the object that it is carrying, it will more likely drop the object at its current location [31].

With the model of Lumer et al in mind, Chen et al [8] proposed a model for ant clustering, where, instead of having both ants and objects on the grid, the objects are represented as different types of ants. Their model is known as the Ant Sleeping Model or ASM. Each ant moves around on a grid until it finds a neighbourhood which is populated by ants of the same kind. An ant is more likely to sleep at a location which contains a higher density of ants of the same kind. If the neighbourhood of an ant (that is currently sleeping) becomes less populated by similar ants, the ant will have a higher probability of waking up and relocating to a different location on the grid. Chen et al proposed the Adaptive Artificial Ant Clustering Algorithm or A^{4}C that performs the ant clustering on a data set.

For the ant clustering CA, both the LF algorithm and the A^{4}C will be used.

4.1 Rules

4.1.1 LF algorithm

The rules of the LF algorithm are based on an ant picking up an object with a pick-up probability of P_p, or an ant dropping an object with a drop probability of P_d. The pick-up probability for an object i is defined as

$$P_p(i) = (\frac{k_p}{k_p + f(i)})^2, \tag{4.1}$$

where k_p is a constant real value that regulates the pick-up sensitivity of objects; a higher value translates into a higher likelihood of picking up similar objects [41]. The function $f(i)$ represents the fitness of the object i (see Equation 4.3). The

Stellenbosch University https://scholar.sun.ac.za

drop probability for an object i is defined as

$$P_d(i) = \begin{cases} 2f(i) & \text{if } f(i) < k_d \\ 1 & \text{otherwise,} \end{cases} \tag{4.2}$$

where k_d is a constant real value that regulates the drop probability; a higher value translates into a need for more similar objects in the neighbourhood to drop an object [41]. The function $f(i)$ is a local estimation of the density of objects and their similarity to the object i, known as the fitness value. It is defined as

$$f(i) = \begin{cases} \frac{1}{N}\sum\limits_{j=1}^{N}(1 - \frac{d(i,j)}{\alpha}) & \text{if } f(i) > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{4.3}$$

where $\frac{1}{N}$ is the normalizing term, and N is equal to the number of cells in the local neighbourhood of object i. The function $d(i,j)$ measures the dissimilarity of objects i and j. More specifically, the Euclidean distance of the value space of objects i and object j is measured. The value space of an object can differ depending on the data being represented by the objects. For the ant clustering CA experiments, one-dimensional data objects will be used; thus an object is represented by a single real value. If the value for object i is x and the value for object j is y, then $d(i,j) = |x - y|$.

Finally, α is the average dissimilarity constant between all objects in the grid, and is predetermined as

$$\alpha = \frac{1}{o_n \times (o_n - 1)} \sum_{i=1}^{o_n}\sum_{j=1}^{o_n} d(i,j), \tag{4.4}$$

where o_n is the number of objects on the CA grid. Note that α is a real value.

In order for an ant to pick up an object, or to drop the object that it is carrying, the fitness value of the object $f(i)$, must first be calculated. Next, either the pick-up or the drop probability for the object is calculated. The relevant probability is measured against a pseudo-random number to determine whether to pick up the object, or to drop the object. After this process, the algorithm finishes by moving the ant to a random neighbouring location. To calculate one generation of the CA, this process is performed for each ant.

4.1.2 A^{4}C

The rules of the A^{4}C is based on whether an ant will sleep at its current location, or wake up and move to a neighbouring location. This process is started by first

Stellenbosch University https://scholar.sun.ac.za

determining the fitness value $f(i)$ of an ant i, using Equation 4.3. With the fitness value calculated, the activation probability P_a of an ant i is calculated as follows:

$$P_a(i) = \frac{\beta^\lambda}{\beta^\lambda + f(i)^\lambda}, \tag{4.5}$$

where both β and λ are constant real values. In this function β represents the threshold of an agent's active fitness, while λ is normally equal to 2 [8]. If $\beta >> P_a(i)$ then ant i has a higher probability of waking up. The activation probability of the ant is measured against a pseudo-random number to determine whether the ant will stay at its current location or move to a neighbouring location. If an ant does move, the location it moves to is determined by calculating which of its neighbouring positions is the most fitting. For this process, the fitness of each of the vacant neighbouring cells are calculated, and the ant will move to the location with the highest fitness. The next generation of the CA is calculated once all the ants have been processed.

4.2 Generation calculation analysis

In this subsection the operations performed to calculate a generation of ant clustering is discussed for both the LF algorithm and the A^{4}C.

4.2.1 LF algorithm

In order to calculate the next generation of the ant clustering CA using the LF algorithm, two steps are performed. The first step involves checking if an ant is carrying an object and if the current location of the ant is vacant, or if the ant is not carrying an object and whether the current location of the ant does contain an object. In either case, two conditional checks are performed. If none of the cases are met, the second step of the algorithm is executed. Otherwise, the first step continues. Next, the fitness value of the object is calculated which involves performing Equation 4.3. For Equation 4.3, a Moore neighbourhood is used; consisting of nine cells. For each cell, a conditional check is performed to determine whether the cell contains an object. If the cell does contain an object, the dissimilarity between the neighbouring object j and object i is calculated, which requires two floating point operations. Next, the quotient of the result and α is calculated and subtracted from 1.0, which is an additional two floating point operations. The result is added to a temporary `float`, since the sum of all dissimilarity values is being calculated. In total, therefore, one conditional check and five floating point operations are performed for each neighbouring object. The sum of the dissimilarity values is divided by the total number of neighbours, which is an additional floating point operation. Finally, a check is performed to determine if the final result is greater than zero.

Stellenbosch University https://scholar.sun.ac.za

In general, a total of 41 floating point operations and nine conditional checks are performed to calculate the fitness value of an object i.

With the fitness value calculated, either the pick-up probability or the drop probability is calculated. For the pick-up probability, three floating point operations must be performed. For the drop probability, a conditional check and a conditional floating point operation must be performed. Having calculated the appropriate probability, a pseudo-random number is generated and a conditional check is performed to determine whether the probability value is less than the pseudo-random number. If the probability value is less than the pseudo-random number, the state of the ant and the state of the cell where the ant is currently located, is updated.

The second step of the of the LF algorithm involves selecting and moving the ant to a random neighbouring location. This process involves generating a pseudo-random number, and depending on the number generated, the ant is moved to one of the eight neighbouring locations, if the location is vacant. This process performs a maximum of eight conditional checks and updates the X and Y coordinate values of the ant.

In the worst case scenario, if all neighbouring cells around an ant contain an object, a total of 41 floating point operations are performed (all during the fitness calculation) and 17 conditional checks (nine during the fitness calculation and eight during the move calculation) are performed. However, since the number of ants are generally far less than the total number of cells on the grid, the total number of operations that are performed is far less than for the Game of Life CA or the clay deformation CA. Typically, the number of objects located on the grid is an order of magnitude less than the number of cells, and the number of ants is an order of magnitude less than the number of objects [31].

4.2.2 A^{4}C

Calculating the next generation of an ant clustering CA with the A^{4}C requires iterating over each ant. For each ant, the fitness of the ant must be calculated with Equation 4.3; as is done with the LF algorithm. As mentioned in Section 4.2.1, for each cell in the neighbourhood of ant i, a conditional check is performed to determine if the cell is an ant, and if so, an additional five floating points operations are performed per ant. In the worst case scenario, if all neighbouring cells contain an ant, 41 floating points operations and nine conditional checks are performed to calculate the fitness value of ant i.

Once the fitness value of an ant has been calculated, the activation probability of the ant must be calculated. This process involves performing five floating point operations. The activation probability value is measured against a pseudo-random number using a conditional check. If the activation-probability is less than the random number, the ant must wake up and relocate; else, the ant stays at its current location.

Stellenbosch University https://scholar.sun.ac.za

If an ant has to relocate, a greedy relocation algorithm is used to determine what the best location is for the ant. This process involves checking the fitness value of the vacant neighbouring cells of the ant, and an additional conditional check is performed to determine if the fitness value of a vacant cell is higher than the previous highest fitness value. If more neighbouring cells are vacant, more operations will have to be performed. In the worst case scenario, where all neighbouring cells are vacant but are surrounded by an outer ring of ants, the fitness value of nine cells must be calculated which involves calculating the dissimilarity values of 40 ants; see Figure 4.22. In total, 205 floating point operations are performed; five for each

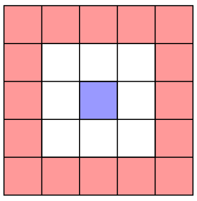


Figure 4.22: Worst case scenario of calculations for a single ant with the A^{4}C. The blue cell is the ant in question, the white cells are its neighbouring cells, and the red cells are the ants surrounding the neighbouring cells.

ant when calculating the dissimilarity values during the fitness value calculations, and the five operations during the activation probability calculation. The total number of conditional checks is 49; one for each ant during the fitness value calculations, one for the activation probability calculation, and eight to determine the best fitness value during the move calculation.

For the A^{4}C, the total number of operations performed per ant, is more than for each cell for the Game of Life CA and the clay deformation CA. However, as mentioned, the number of ants on which the work is performed, is one order of magnitude less than the number of cells in the CA grid.

4.2.3 Ant clustering CA time complexity

For both of the LF algorithm and the A^{4}C, the number of ants is dependent on the total number of cells. The number of ants is determined by multiplying an ant density value a with the number of cells, where $0 < a \leq 1$. The number of ants is equal to aN^2. Thus, the time complexity of an ant clustering CA is $O(N^2)$ since the calculations performed per ant is done in constant time.

4.3 Experimental setup

In this subsection the experimental setup of both the the LF algorithm and the A^{4}C are discussed.

Stellenbosch University https://scholar.sun.ac.za

4.3.1 LF algorithm

For the LF algorithm implementations (both sequential and parallel), the experiments are performed on a fixed grid size while changing the ant density value. A grid size of 100×100 is used, which is a total of 10 000 cells. The number of objects is one order of magnitude less than the number of cells, giving 1000 objects. The number of ants, or the ant density, varies as indicated in Table 4.15, where the number of ants is equal to a density value multiplied by the number of objects. In order to test the scalability of the algorithm, a second set of experiments are performed on a grid size of 200×200, which contains 2000 objects. The same ant density values indicated in Table 4.15 are used.

Table 4.15: Ant density values for LF algorithm.

Number of cells: 100^2		
Experiment	Ant density	Number of ants
1	1.00	1000
2	0.75	750
3	0.50	500
4	0.25	250
5	0.10	100

For each experiment the total time to calculate 10 000 generations is measured in milliseconds. This process is repeated ten times to validate the calculation times measured.

4.3.2 A^{4}C

For the A^{4}C, objects are portrayed as ants, and thus the number of ants to iterate over is typically more than for the LF algorithm. For the A^{4}C algorithm implementations (both sequential and parallel) a 100×100 grid is used for the first set of experiments, where the number of ants is equal to an order of magnitude less than the number of cells, multiplied by an ant density value. The ant density values are given in Table 4.16. The second set of experiments will be performed on a grid size

Table 4.16: Ant density values for A^{4}C method.

Number of cells: 100^2		
Experiment	Ant density	Number of ants
1	0.20	2000
2	0.10	1000
3	0.05	500

Stellenbosch University https://scholar.sun.ac.za

of 200×200 to test the scalability of the A^{4}C, while using the same ant density values. For each experiment the total time to calculate 10 000 generations is measured in milliseconds. This process is repeated ten times to validate the calculation times measured.

4.4 Sequential implementation: LF algorithm

In this subsection the sequential implementation of the LF algorithm for the ant clustering CA is considered.

4.4.1 LF algorithm sequential code extracts

The sequential implementation for the LF algorithm is encompassed in the function `gridNextStateSEQ()` of the derived ant clustering `AntLFA` class in Listing B.12. This function iterates over every ant and performs the rule set of the LF algorithm as discussed in Section 4.2.1. Listing B.13 provides the function that is used to calculate the fitness value of an ant. To pick up an object the pick-up probability must be calculated (see Listing B.14). Else, to drop an object the drop probability must be calculated (see Listing B.15).

4.4.2 LF algorithm sequential experiment results

The results for the sequential LF algorithm implementation are given in Table 4.17, which provides the average calculation times and the average number of CA generations calculated per second, for both Computer A and Computer B. Figure 4.23 shows the calculation time series.
The results in Table 4.17 indicate that a higher ant density results in a longer calculation time, since there are more ants to process. However, a higher ant density results in a better overall clustering of objects. Figure 4.24 shows a typical unclustered grid and Figure 4.25 shows the clustering for each ant density value, after 10 000 generations.

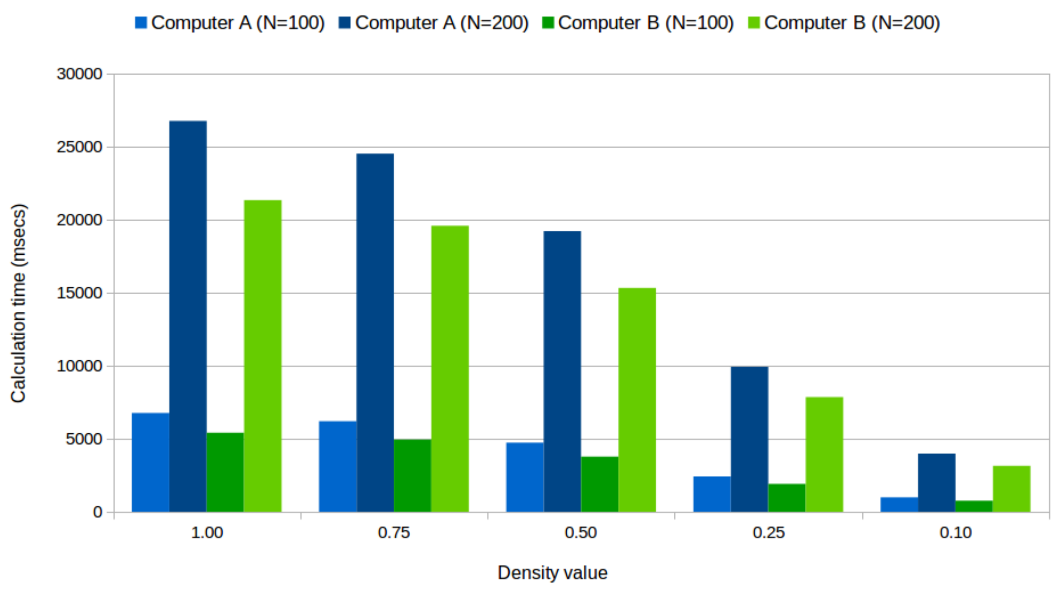
Notice that the calculation times for $N = 200$, for each density value, is more or less four times greater than the calculation times for $N = 100$ (see also Figure 4.23). The number of objects is ten times less than the number of cells (one order of magnitude less). Thus, for an $N \times N$ size grid, the number of objects I, is equal to $\frac{1}{10}N^2$. The number of ants is equal to a fraction of the number of objects (as indicated in Table 4.15), and thus for I objects the number of ants is equal to $I \times r$, where $0 < r \leq 1$. Since $I = \frac{1}{10}N^2$, the number of ants is equal to $\frac{1}{10}rN^2$. If the time taken to process one ant is t, the calculation time for one generation is equal to $\frac{1}{10}rN^2t$. The dominating factor in this formula is N^2. A quadratic increase in the calculation time is therefore expected as the value of N increases, which is indicated by the results.

Stellenbosch University https://scholar.sun.ac.za

Table 4.17: Results for ant clustering LF algorithm sequential experiments.

	Calculation time (msecs)		Generations per second	
Ants	**Computer A**	**Computer B**	**Computer A**	**Computer B**
100^2 **cells;** 1000 **objects**				
1000	6771	5417	1477	1846
750	6208	4945	1611	2022
500	4737	3781	2111	2645
250	2425	1914	4123	5225
100	988	762	10126	13129
200^2 **cells;** 4000 **objects**				
4000	26792	21363	373	468
3000	24545	19605	407	510
2000	19239	15345	520	652
1000	9936	7862	1006	1272
400	3986	3146	2509	3179

Figure 4.23: LF algorithm sequential calculation times.



4.5 Sequential implementation: A^{4}C

In this subsection the sequential implementation of the A^{4}C for the ant clustering CA is considered.

Stellenbosch University https://scholar.sun.ac.za

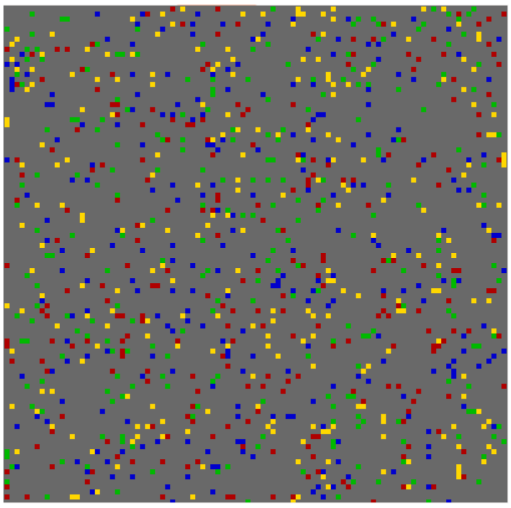


Figure 4.24: Objects randomly distributed on a 100 × 100 size grid.

4.5.1 A^{4}C sequential code extracts

The sequential implementation for the A^{4}C is contained in the `gridNextStateSEQ()` function of the derived ant clustering `Ant` class in Listing B.16.

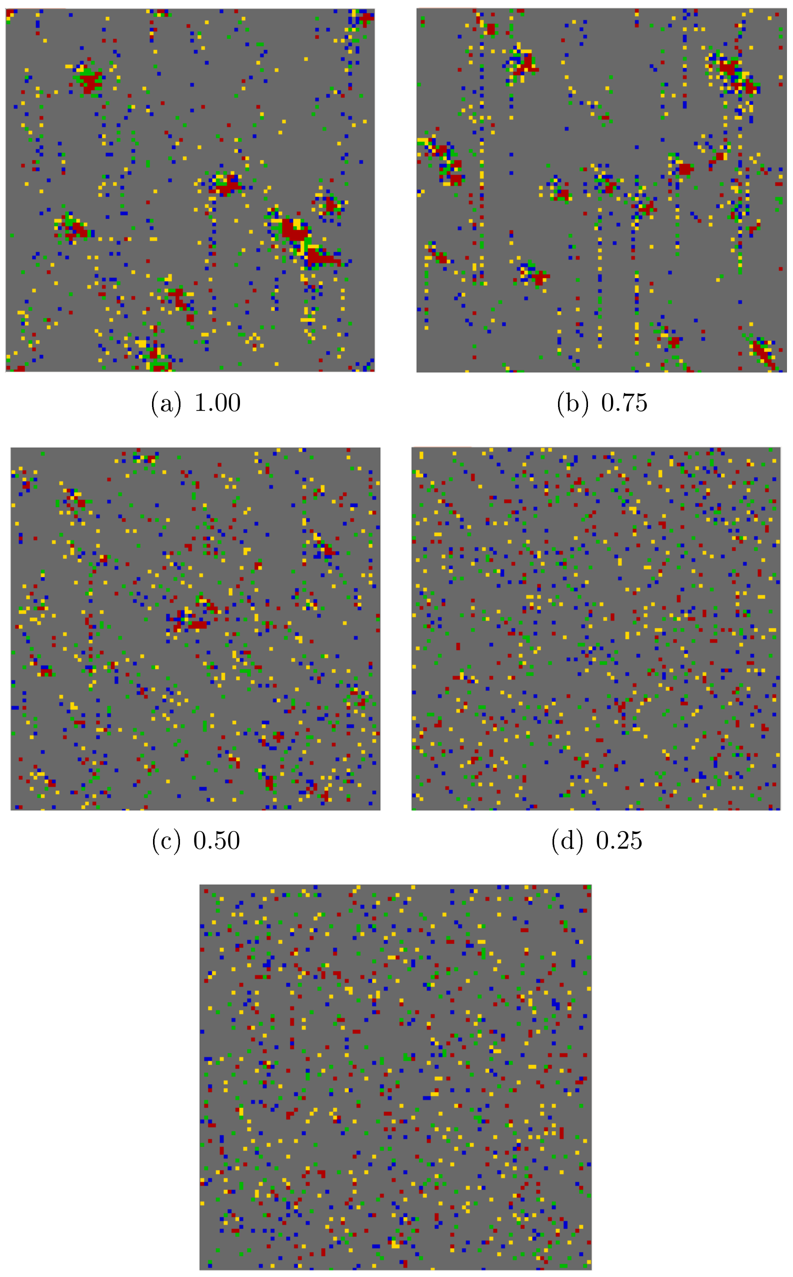
This function iterates over every ant and performs the rule set of the A^{4}C as discussed in Section 4.2.2. Listing B.13 provides the function that is used to calculate the ant fitness value. The fitness value is used to calculate the activation probability of an ant (see Listing B.17). Listing B.18 gives the function that is used to determine the location that an ant moves to.

4.5.2 A^{4}C sequential experiment results

Table 4.18: Results for ant clustering A^{4}C sequential experiments.

	Calculation time (msecs)		**Generations per second**	
Ants	**Computer A**	**Computer B**	**Computer A**	**Computer B**
100^2 **cells**				
2000	8141	9946	1228	1005
1000	4133	4996	2420	2002
500	2169	2553	4611	3917
200^2 **cells**				
8000	59438	54858	168	182
4000	16825	20206	594	495
2000	8717	10281	1147	973

Stellenbosch University https://scholar.sun.ac.za



(a) 1.00

(b) 0.75

(c) 0.50

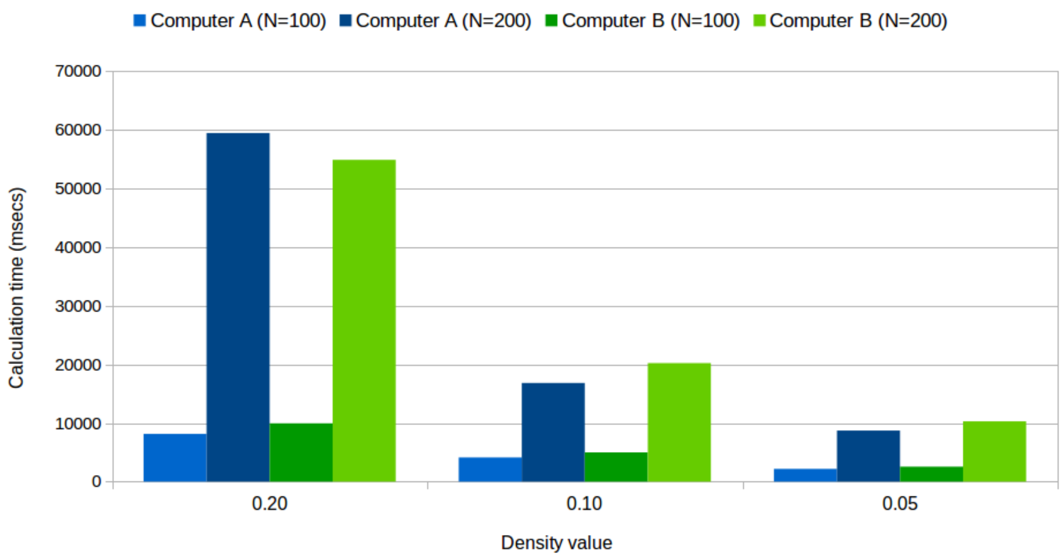
(d) 0.25

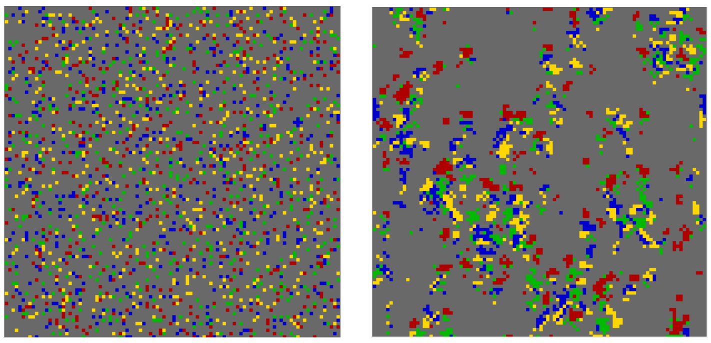
(e) 0.10

Figure 4.25: LF algorithm ant clustering for different ant density values.

Stellenbosch University https://scholar.sun.ac.za

Figure 4.26: A^{4}C sequential calculation times.





(a) Unclustered (b) Clustered

Figure 4.27: A^{4}C implementation for a density value of 0.2.

The results for the sequential A^{4}C implementation for ant clustering is given in Table 4.18. Figure 4.26 shows the calculation time series.

As with the LF algorithm, the ant density values for the A^{4}C influences the calculation times. However, the density of the ants does not influence the overall clustering of the ants after 10 000 generations. Figures 4.27, 4.28, and 4.29 show the clustering of each ant density value for a grid size of 100 $\times$ 100, after 10 000 generations.

The A^{4}C also performs better clustering than the LF algorithm, in a shorter amount of time. For example, the LF algorithm performed on a 100 $\times$ 100 size grid

Stellenbosch University https://scholar.sun.ac.za

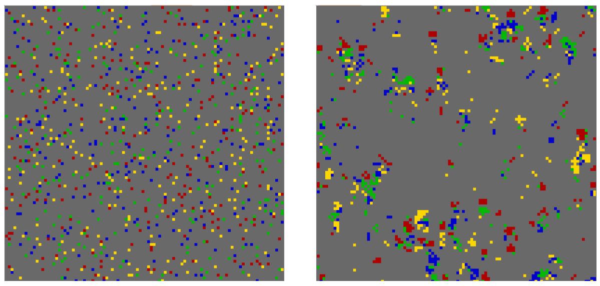

(a) Unclustered (b) Clustered

Figure 4.28: A^{4}C implementation for a density value of 0.1.

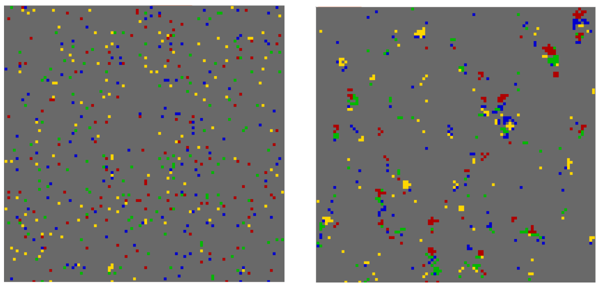

(a) Unclustered (b) Clustered

Figure 4.29: A^{4}C implementation for a density value of 0.05.

for 1000 ants for 10 000 generations, is slower than the A^{4}C for the same grid size, number of ants and number of generations, by 2.64 seconds on Computer A and 0.42 seconds on Computer B. The A^{4}C also performs better clustering since the A^{4}C has less disjoint objects; refer to Figure 4.30.

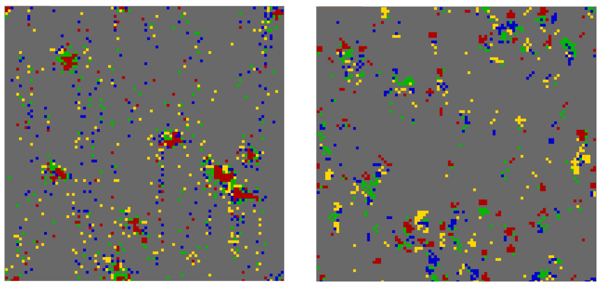

(a) LF algorithm (b) A^{4}C method

Figure 4.30: LF algorithm versus the A^{4}C, for the same grid size and number of ants.

The calculation time of the A^{4}C tends to grow quadratically as the value of

Stellenbosch University https://scholar.sun.ac.za

N increases, and consequently as the number of ants increases. As with the LF algorithm, the A^{4}C has $O(N^2)$ number of ants to process, and therefore has a $O(N^2)$ time complexity, since the time to process one ant is equal to a constant time t (see Figure 4.26).

4.6 Parallel implementation: LF algorithm

In this subsection the parallel implementations of the LF algorithm for the ant clustering CA are considered.

4.6.1 LF algorithm parallel code extracts

For the ant clustering CA ants move around on the grid. If ants are processed sequentially, no two ants will be able to move to the same location when calculating the next generation of the CA, since each ant that has been moved will be directly updated on the grid. Therefore, when processing the following ant, the previous ant is already at its new location. However, if the process of moving ants is done in parallel, collisions can occur when two or more ants attempt to move to the same cell simultaneously. To compensate for collisions, two parallel algorithms are proposed.

The first parallel algorithm is a sequential-parallel hybrid algorithm (henceforth known as a hybrid parallel algorithm). The first part of the LF algorithm which calculates the appropriate probabilities for the ants, are performed by the GPU. Since the ants are not being moved, no collisions can occur. The CPU performs the second part of the algorithm sequentially. This algorithm is encompassed in the function `gridNextStatePARhybrid()` of the derived ant clustering `AntLFA` class in Listing B.19.

The first part of the algorithm, to be processed by the GPU, involves the calculation of the fitness value of an ant, and either the pick-up probability or the drop probability. The function `Ant_Fitness_ApT()` calculates the fitness value of each ant; see Listing B.20. The calculated value is then sent to the function `Ant_LFAprob_ApT()` which calculates the appropriate probability for the particular ant assigned to a thread; see Listing B.21. The calculated probabilities are sent to main memory. This process is performed for each generation.

Once the calculated probabilities are transferred from GPU memory to main memory, the sequential part of the algorithm continues. In the sequential part of the algorithm, an ant will either pick up an object or drop an object, after which the ant moves to a new location.

The second parallel algorithm is performed exclusively on the GPU and is performed with the function `Ant_Move_ApT_LFA()`. This function is identical to the function given in Listing B.12. However, a CUDA `__syncthread()` operation is performed

Stellenbosch University https://scholar.sun.ac.za

which acts as a barrier to synchronize all threads before the ants are moved. In order to avoid collisions, each thread checks if any other thread with a lower thread ID wants to move its assigned ant to the same location, in which case the current thread will not move its ant. Thus, threads with a lower thread ID have a higher precedence.

Since the rule application function is only applied to the ants and not to each cell on the grid (as is done with the Game of Life and clay deformation CA), the grid-per-thread and row-per-thread segmentation methods are not applicable for the ant clustering CA. Both parallel algorithms only use an ant-per-thread segmentation method, which is equivalent to a cell-per-thread segmentation method.

4.6.2 LF algorithm parallel experiment results

The results for the hybrid parallel LF algorithm implementation are given in Table 4.19. Figure 4.31 shows the calculation time series.

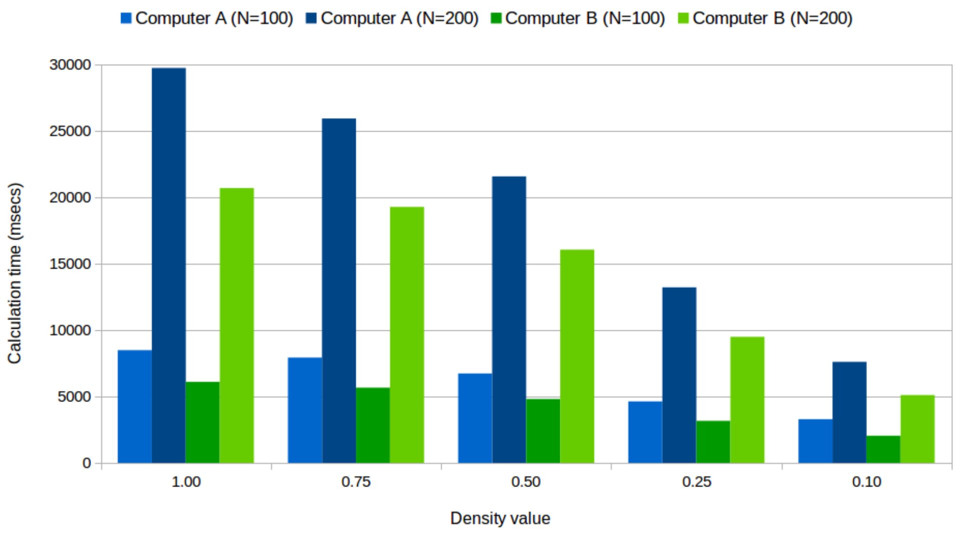
Table 4.19: Results for ant clustering hybrid parallel LF algorithm experiments.

	Calculation time (msecs)		**Generations per second**	
Ants	**Computer A**	**Computer B**	**Computer A**	**Computer B**
100^2 **cells;** 1000 **objects**				
1000	8498	6110	1177	1637
750	7938	5673	1260	1763
500	6739	4820	1484	2075
250	4632	3166	2159	3158
100	3295	2046	3035	4887
200^2 **cells;** 4000 **objects**				
4000	29764	20714	336	483
3000	25959	19296	385	518
2000	21598	16075	463	622
1000	13230	9504	756	1052
400	7619	5116	1313	1955

When analyzing the calculation times of the hybrid parallel algorithm, notice that the calculation times for $N = 200$, with higher ant density values, tend to be closer to four times the calculation time for $N = 100$, with the same ant density values. For $N = 200$ with lower ant density values, there are less ants to iterate over, and the calculation times are far less than the expected value of four times the calculation time for $N = 100$, with the same density values; refer to Figure 4.31. This is because the most computationally expensive part of the algorithm is performed in parallel, while the sequential part of the algorithm will only conditionally

Stellenbosch University https://scholar.sun.ac.za

Figure 4.31: Hybrid parallel LF algorithm calculation times.



pick up or drop objects and move ants to new locations. Thus, if the number of ants is less than or equal to the number of GPU processing cores, the probabilities will be calculated simultaneously. If there are more ants than the number of GPU processing cores, each core will start processing more ants, thus leading to an increase in the calculation time constant.

The time complexity of the ant clustering CA is $O(N^2)$ as discussed in Section 4.2.3. Since the hybrid parallel implementation processes all the ants sequentially in the second part of the algorithm and since the GPU processing resources stay constant, the growth in calculation time is expected to be quadratic as the value of N increases.

The results for the fully parallel LF algorithm implementation are given in Table 4.20. The time taken to calculate 10 000 generations is significantly longer than the sequential or hybrid parallel implementations. This is because threads with a higher thread ID must scan through a larger portion of the ants, to check whether other ants want to move to the same location. This measure to prevent collisions, along with the random number generation performed on the GPU, slows the fully parallel LF algorithm down significantly. Another problem that occurs because of the measure to prevent collisions is that the clustering of objects, for a higher density of ants, is also negatively impacted. Since there is a higher probability that two or more ants want to move to the same location, less ants will ultimately be allowed to move on the grid; refer to Figure 4.32

A lower ant density results in a faster calculation time as is expected, although the difference in calculation times is small. Notice that the calculation time for

Stellenbosch University https://scholar.sun.ac.za

Table 4.20: Results for ant clustering fully parallel LF algorithm experiments.

	Calculation time (msecs)		**Generations per second**	
Ants	**Computer A**	**Computer B**	**Computer A**	**Computer B**
100^2 **cells;** 1000 **objects**				
1000	129219	60278	77	166
750	116732	55659	86	180
500	112431	51969	89	192
250	98580	45719	101	219
100	72427	33682	138	297
200^2 **cells;** 4000 **objects**				
4000	283316	96812	35	103
3000	176143	86814	57	115
2000	151944	71703	66	139
1000	128505	60135	78	166
400	100891	49033	99	204

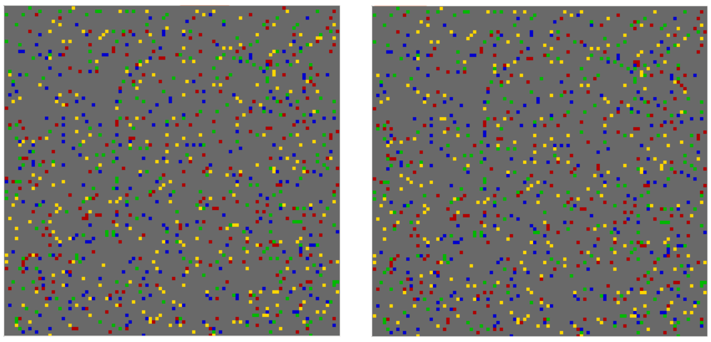

(a) Unclustered (b) Clustered

Figure 4.32: Fully parallel LF algorithm performed with an ant density of 1.00 on a 100×100 size grid.

$N = 200$ is between 1.32 and 1.61 times the calculation time for $N = 100$. A quadratic time increase is expected since the time complexity is $O(N^2)$, but for smaller values of N, more ants are processed simultaneously. As N increases, the calculation time will tend towards a quadratic time increase, since the number of ants processed simultaneously stays constant, while the number of ants to be processed increases quadratically.

Stellenbosch University https://scholar.sun.ac.za

4.7 Parallel implementations: A^{4}C

In this subsection the parallel implementations of the A^{4}C for the ant clustering CA are considered.

4.7.1 A^{4}C parallel code extracts

For the A^{4}C, two parallel algorithms are also proposed to process the CA, which are similar to the parallel algorithms used for the LF algorithm. The first parallel algorithm is also a hybrid parallel algorithm, encompassed in the function `gridNextStatePARhybrid()` of the derived `Ant` class in Listing B.22. The part of the algorithm performed on the GPU calculates the fitness of each ant; see Listing B.20. The activation probability of an ant is calculated with the function `Ant_ActPrb_ApT()`; see Listing B.23.

Once the activation probabilities have been calculated and the data is sent back to the main memory, the sequential part of the algorithm continues to process each ant to determine whether an ant must sleep, or wake up and move.

The second parallel algorithm is performed exclusively on the GPU and is also a fully parallel algorithm. The fully parallel A^{4}C is identical to the function `Ant_Move_ApT()` given in Listing B.16. As with the fully parallel LF algorithm, collisions need to be eliminated. Thus, each thread checks if a thread with a lower thread ID wants to move its ant to the same location, in which case the current thread will not move its ant.

The parallel A^{4}C algorithms also use the ant-per-thread segmentation method.

4.7.2 A^{4}C parallel experiment results

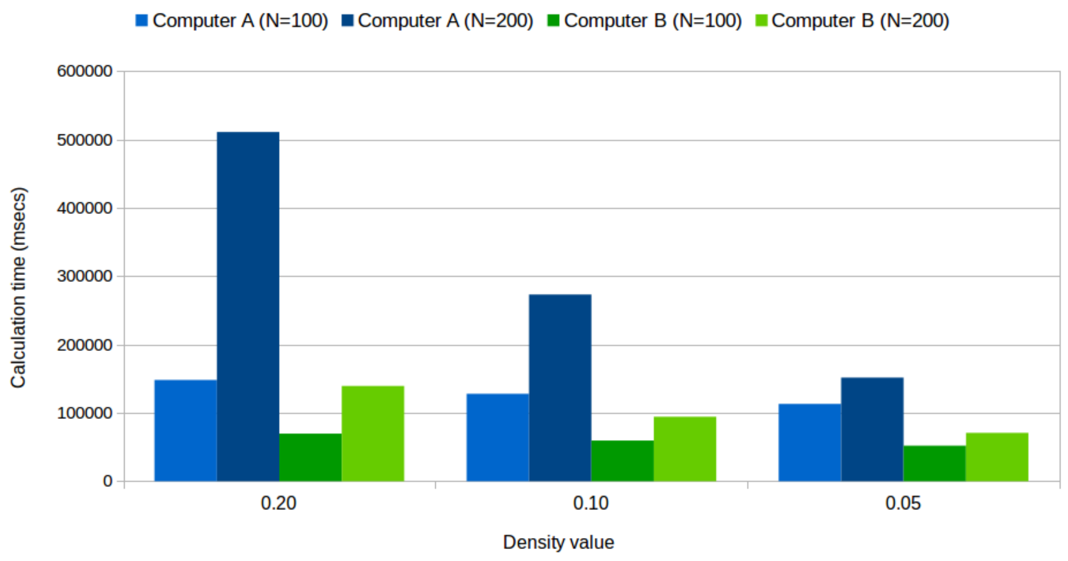
The results for the hybrid parallel A^{4}C implementation are given in Table 4.21. The hybrid parallel method only calculates the activation probability of each ant on the GPU, while the locations that the ants are moved to are calculated in the sequential part of algorithm. Therefore the ant clustering performed by the hybrid parallel implementation does not differ from the sequential A^{4}C implementation; only the calculation times do.
The calculation times differ depending on the number of ants, as expected. However, the difference in calculation times are smaller than for the sequential A^{4}C implementation. For a smaller number of ants (for $N = 100$), the calculation times tend to be closer. For a larger number of ants (for $N = 200$), the difference in calculation times is greater. For a smaller number of ants, more activation probability values are simultaneously calculated, than for a larger number of ants; refer to Figure 4.33.

Stellenbosch University https://scholar.sun.ac.za

Table 4.21: Results for ant clustering hybrid parallel A^4C experiments.

	Calculation time (msecs)		**Generations per second**	
Ants	**Computer A**	**Computer B**	**Computer A**	**Computer B**
100^2 **cells**				
2000	4679	2899	2137	3450
1000	3948	2385	2533	4192
500	3600	2153	2778	4646
200^2 **cells**				
8000	36588	24923	273	401
4000	10395	5519	962	1812
2000	7085	4468	1411	2238

Figure 4.33: Hybrid parallel A^4C calculation times.



The time complexity of the hybrid parallel function is the same as the sequential A^4C implementation. Although the first part of the hybrid parallel function processes ants simultaneously, the number of ants that can be processed is still a fixed number, which depends on the number of processing cores on the GPU. Thus, for larger values of N, the calculation times of the hybrid parallel function will grow quadratically, since the time complexity is $O(N^2)$.

The results for the fully parallel A^4C implementation are given in Table 4.22. As with the fully parallel LF algorithm implementation, the calculation times for the

Stellenbosch University https://scholar.sun.ac.za

fully parallel A^{4}C implementation are far greater than the sequential or hybrid parallel implementations.

Table 4.22: Results for ant clustering fully parallel A^{4}C experiments.

	Calculation time (msecs)		**Generations per second**	
Ants	**Computer A**	**Computer B**	**Computer A**	**Computer B**
100^2 **cells**				
2000	148063	69583	68	144
1000	127855	59550	78	168
500	113089	51941	88	193
200^2 **cells**				
8000	510765	139303	20	72
4000	273069	94351	37	106
2000	151592	70660	66	142

Because of the need to compensate for potential collisions, and since random numbers (needed to measure the activation probabilities against) are generated on the GPU, the fully parallel A^{4}C implementation does not perform well when measured against the hybrid parallel implementation.

The fully parallel A^{4}C implementation does however perform better clustering than the fully parallel LF algorithm implementation. The A^{4}C implementation is 1.36 seconds faster the LF algorithm implementation, for 1000 ants on a 100×100 size grid; refer to Figure 4.34.

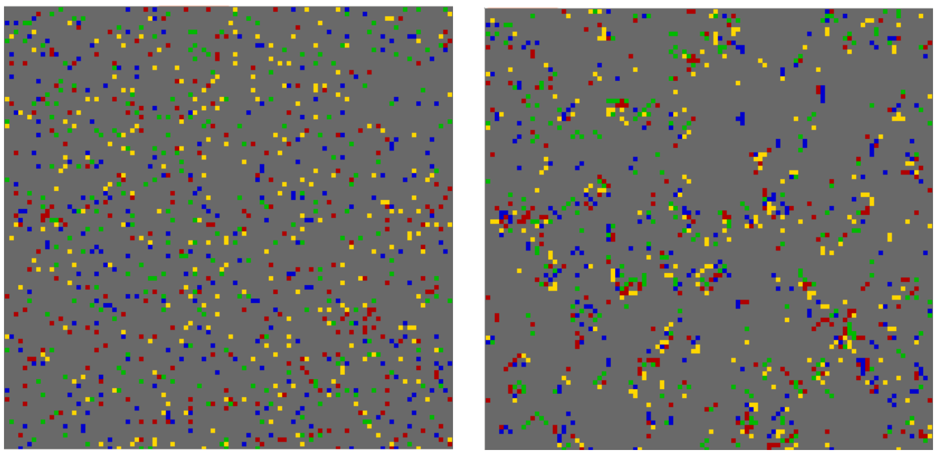

(a) Unclustered (b) Clustered

Figure 4.34: Fully parallel A^{4}C implementation performed with an ant density of 0.10 on a 100×100 size grid.

Stellenbosch University https://scholar.sun.ac.za

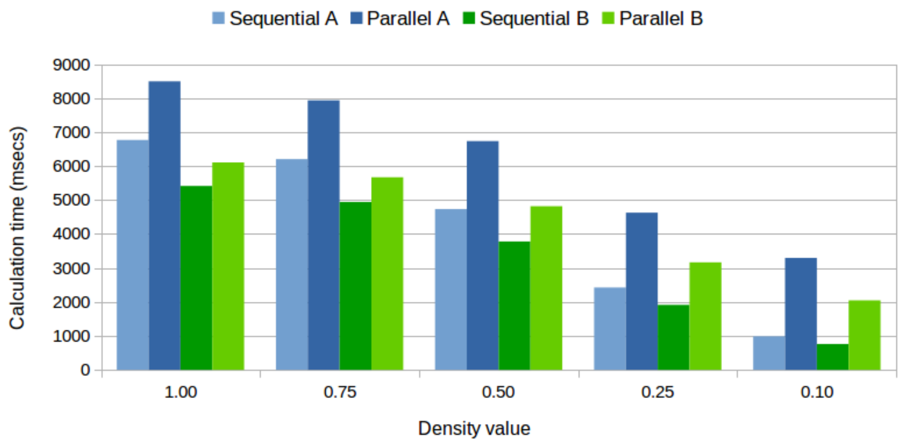
As with the hybrid parallel A^{4}C implementation, the time complexity of the fully parallel A^{4}C implementation is also $O(N^2)$, and the number of ants processed simultaneously stays constant. Thus, as N increases, the calculation time of the fully parallel A^{4}C implementation will tend towards a quadratic time increase.

4.8 Sequential versus parallel experiment data

The sequential implementations for the ant clustering CA tend to perform better than the proposed hybrid parallel implementations for both the LF algorithm and the A^{4}C.

The sequential LF algorithm implementation outperforms the hybrid parallel LF algorithm implementation for all the experiments performed. However, as the number of ants increases, the calculation time of the hybrid parallel implementation is closer to the sequential implementation; see Figure 4.35 and Figure 4.36. The overhead of transferring the fitness values, calculated by the GPU, back to

Figure 4.35: LF algorithm: sequential and hybrid parallel calculation times for $N = 100$.



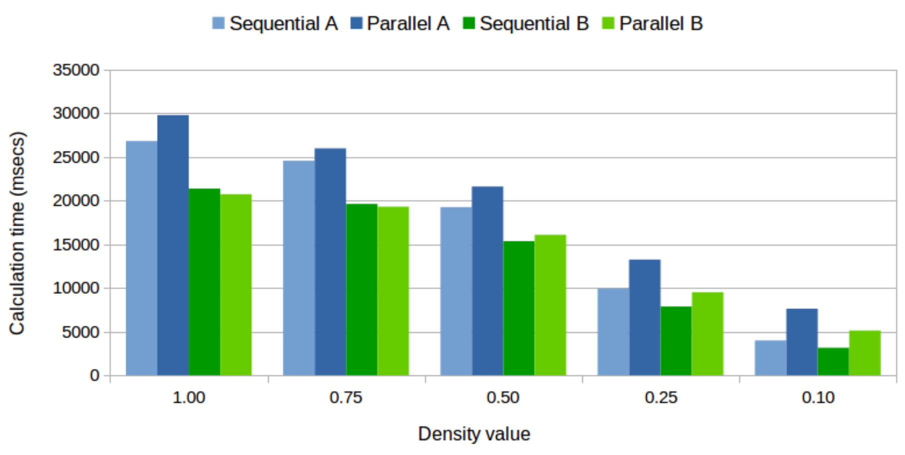
main memory hampers the overall performance of the hybrid parallel LF algorithm implementation.

The fully parallel LF algorithm is between 7.2 and 73.3 times slower than the sequential algorithm for Computer A, and between 4.4 to 44.2 for Computer B. These values are dependent on the number of ants to be processed, where the sequential implementation is faster for less ants. The fully parallel LF algorithm implementation also delivers poor object clustering, and this algorithm should not be used for any form of ant clustering, since it does not have any benefits.

The sequential A^{4}C implementation only outperforms the hybrid parallel A^{4}C im-

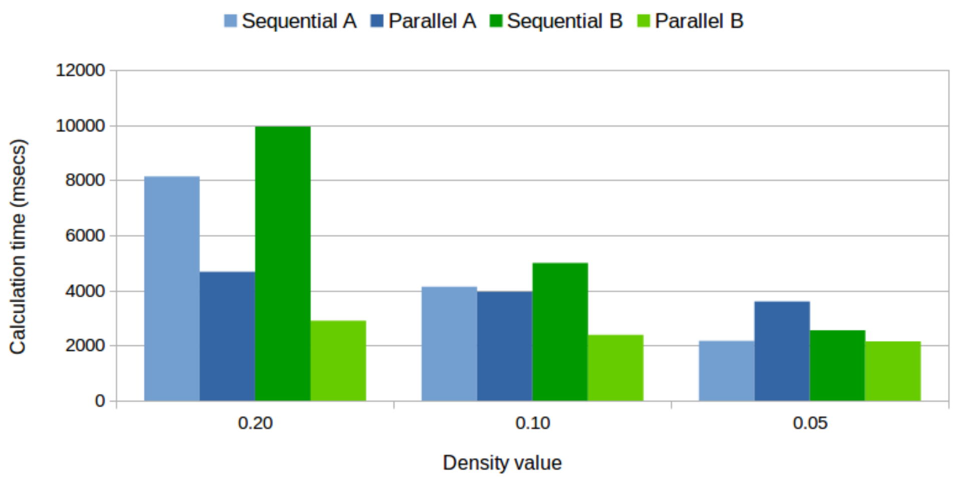
Stellenbosch University https://scholar.sun.ac.za

Figure 4.36: LF algorithm: sequential and hybrid parallel calculation times for $N = 200$.



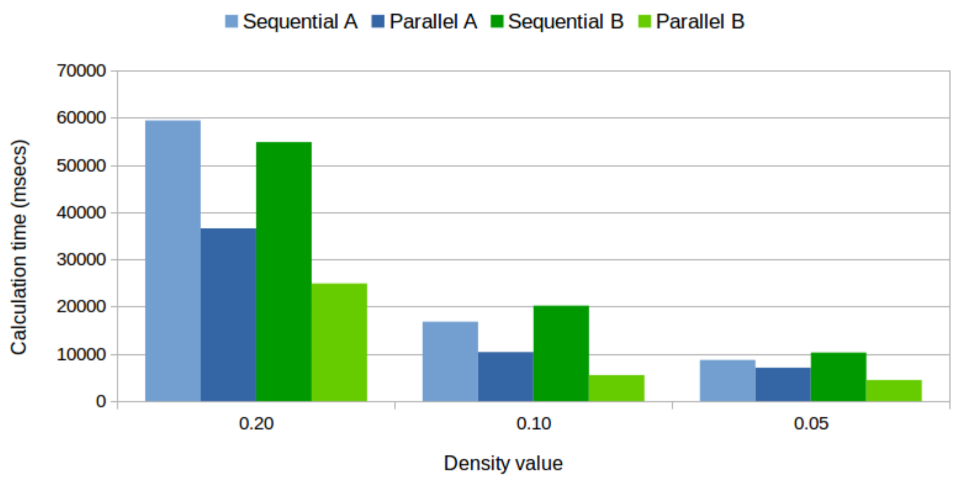
plementation for $N = 100$ and a density value of 0.50, on Computer A. For all the other experiments, the hybrid parallel A^{4}C implementation outperforms the sequential implementation. As with the LF algorithm, the number of ants to be processed influences the calculation times. For a larger number of ants, the hybrid parallel implementation tends to outperform the sequential implementation by a greater margin, whereas if the number of ants is smaller, the sequential implementation calculation times tend to be closer to the hybrid parallel implementation calculation times; see Figure 4.37 and Figure 4.38.

Figure 4.37: A^{4}C: sequential and hybrid parallel calculation times for $N = 100$.



Stellenbosch University https://scholar.sun.ac.za

Figure 4.38: A^{4}C: sequential and hybrid parallel calculation times for $N = 200$.



The fully parallel A^{4}C implementation is outperformed by the sequential A^{4}C implementation. For Computer A, the fully parallel implementation is between 8.6 and 52.1 times slower than the sequential implementation, and between 2.5 and 20.3 for Computer B. Again, the number of ants to be processed has an influence on the difference in calculation times. Although the fully parallel A^{4}C implementation performs better ant clustering than its LF algorithm counterpart, it is still slower than the sequential implementation by a significant margin, and therefore should not be used.

Ultimately, from the experiments performed on the ant clustering CA, it is clear that the A^{4}C is better than the LF algorithm, both in terms of calculation time and object clustering. From the experimental data it is also found that it is better to use a sequential implementation for clustering when the ant density is far lower than the number of cells. However, for a larger grid size and for a higher density of ants, it is better to use the hybrid parallel implementation to perform the clustering, since the influence of the overhead of transferring data to and from the GPU for each generation becomes less.

5 GPU performance

In this section the effectiveness of the GPU is discussed by analyzing the performance of the different segmentation methods performed on the GPU, against the sequential implementations that are performed on the CPU. The results of the different CA are compared according to the potential speed up for the segmentation methods used, and when it is beneficial to use the GPU over a sequential rule

Stellenbosch University https://scholar.sun.ac.za

application function.

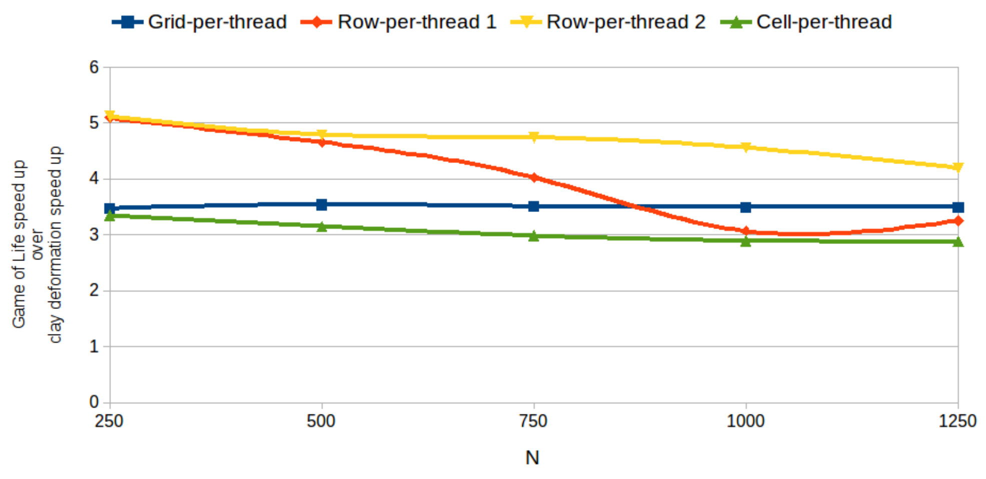
5.1 Comparison of results

For the Game of Life and clay deformation CA, the rule application functions are applied to each cell in the CA grid, whereas for the ant clustering CA the rule application function is only applied to a subset of the CA grid, where the ants are located. For both the Game of Life and clay deformation CA, the grid-per-thread, row-per-thread, and cell-per-thread parallel data segmentation methods are implemented. For the ant clustering CA, an ant-per-thread segmentation method, which is equivalent to the cell-per-thread segmentation method, is implemented.

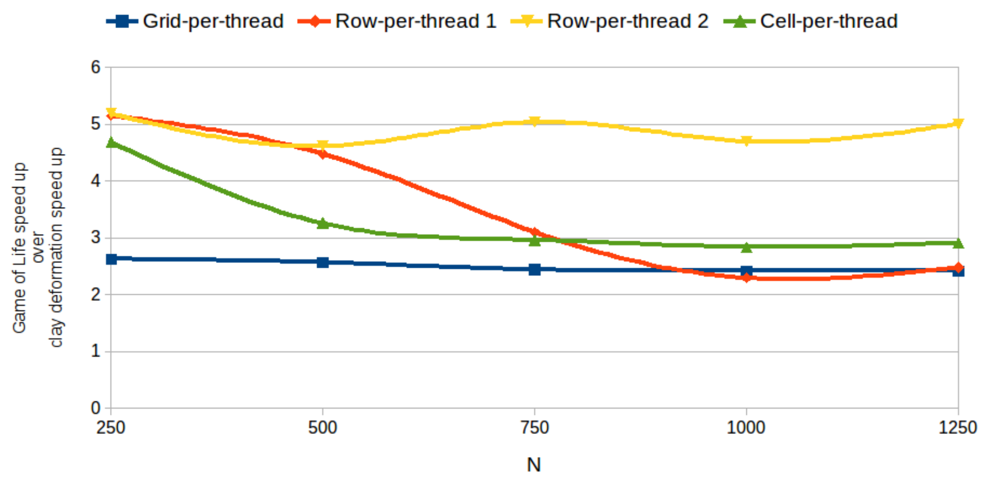
For the grid-per-thread segmentation method, the Game of Life implementation performs better than the clay deformation implementation when comparing the difference between the speed up of the sequential implementations over the parallel implementations. For Computer A, the Game of Life grid-per-thread implementation is between 3 and 4 times more effective than the clay deformation grid-per-thread implementation, and between 2 and 3 times for Computer B. The GPU delivers a better efficiency for rule application functions that only require conditional checks and integer operations, as is the case with the rule application function for the Game of Life CA. For the clay deformation rule application function, where floating point operations are required, the GPU delivers slower performance.
This trend is also seen when comparing the row-per-thread and cell-per-thread segmentation methods for the Game of Life and clay deformation CA; refer to Figure 4.39. The different row-per-thread implementations for the Game of Life CA start to provide a speed up over the sequential implementation for smaller grid sizes than in the case of the clay deformation CA. For the first row-per-thread method, the Game of Life implementation starts to provide a speed up over the sequential implementation for $N \geq 296$ for Computer A, and for $N \geq 211$ for Computer B. For the clay deformation CA, the first row-per-thread implementation does not provide a speed up over the sequential implementation for the experiments performed on Computer A. For Computer B, the first row-per-thread method only starts to provide a speed up for $N \geq 1250$. When comparing the effectiveness of the Game of Life and the clay deformation implementations, the first row-per-thread implementation for Game of Life is between 3 and 5 times more effective than the clay deformation implementation for Computer A, and between 2 and 5 times for Computer B.

In the case of the second row-per-thread method, the Game of Life CA implementation starts to provide a speed up over the sequential implementation for $N \geq 296$ for Computer A, and for $N \geq 136$ for Computer B. In the case of the clay deformation CA, the second row-per-thread implementation does not provide a speed up over the sequential implementation for the experiments performed on

Stellenbosch University https://scholar.sun.ac.za



(a) Computer A



(b) Computer B

Figure 4.39: Parallel segmentation methods: speed ups for the Game of Life CA measured against the speed ups for the clay deformation CA.

Computer A. For Computer B, the second row-per-thread method only starts to provide a speed up for $N \geq 1020$. When comparing the effectiveness of the Game of Life and the clay deformation implementations, the second row-per-thread implementation is between 4 and 5 times more effective than the clay deformation implementation for Computer A, and is about 5 times more effective for Computer B.

For the cell-per-thread implementation for the Game of Life CA, a speed up is

Stellenbosch University https://scholar.sun.ac.za

noticed for smaller grid sizes than for the clay deformation implementation. The cell-per-thread implementation for the Game of Life CA starts to outperform the sequential implementation for $N \geq 13$ for Computer A, and $N \geq 15$ for Computer B. For the clay deformation cell-per-thread implementation, a speed up is only noticed for $N \geq 76$ for Computer A, and $N \geq 64$ for Computer B. When comparing the effectiveness of the Game of Life and the clay deformation implementations, the cell-per-thread implementation is about 3 times more effective than the clay deformation implementation for Computer A, and is between 3 and 5 times for Computer B.

For both the Game of Life and clay deformation CA, the cell-per-thread segmentation method gives the best performance of all four segmentation methods used, since all of the CUDA cores are utilized. Figure 4.40 shows the speed up of the cell-per-thread method over the other segmentation methods.

For the ant clustering CA, the hybrid parallel implementations of the LF algorithm only provides a speed up over the sequential implementation when the number of ants is greater or equal to 3000 on Computer B, where the speed up factor is about 1.02. For Computer A, no speed up is noticed for the experiments performed. For the hybrid parallel implementation of the A^4C, a speed up over the sequential implementation is noticed on Computer A, when the number of ants is greater or equal to 1000. For Computer B, a speed up is noticed when the number of ants is greater or equal to 500. The speed up factor for Computer A is between 1.05 and 1.74, and is between 1.19 and about 4 for Computer B.

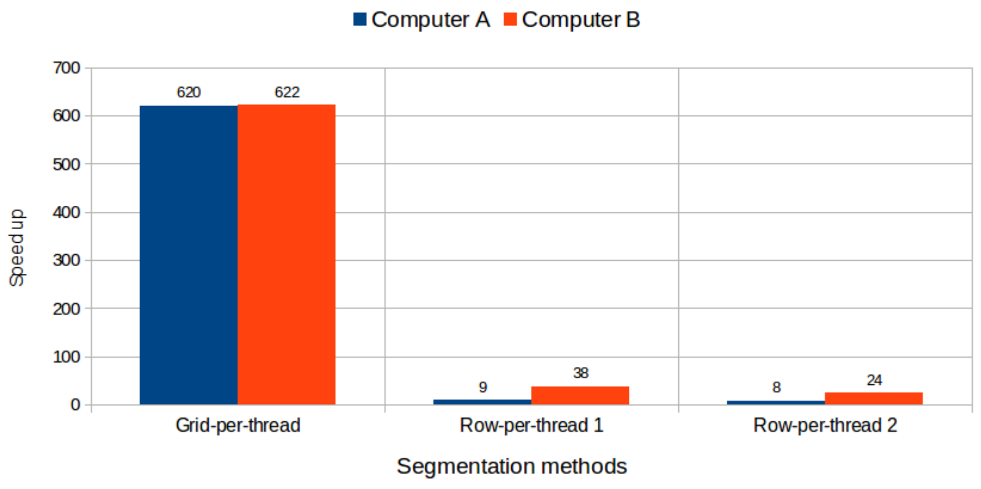
When comparing the hybrid parallel implementations for the ant clustering CA with the cell-per-thread implementations used for both the Game of Life and clay deformation CA, one can see a significant increase in performance gained when the work size is increased. For the ant clustering CA, the hybrid parallel implementations provides a speed up over the sequential implementations when the number of ants pass a certain threshold (depending on the algorithm used) and the speed up factor increases as the number of ants increases. This trend is also noticed for the Game of Life and clay deformation CA.

The fully parallel implementations for the LF algorithm and the A^4C tend to deliver equal performance. However, since the A^4C tends to performs less calculations than the LF algorithm, the sequential implementation of the A^4C outperforms the fully parallel implementation by a greater margin than in the case of the LF algorithm.

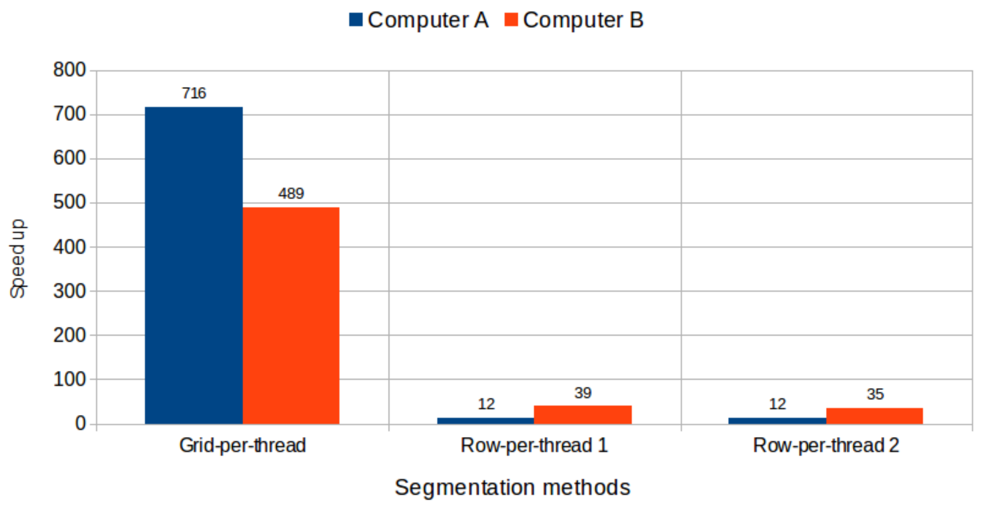
5.2 Parallel over sequential

From the results discussed above, it is clear that a sequential implementation of a CA rule application function can provide better performance than even the fastest of the parallel implementations, the cell-per-thread data segmentation method, but only tends to occur for small grid sizes. Depending on the CA and the com-

Stellenbosch University https://scholar.sun.ac.za



(a) Game of Life



(b) Clay deformation

Figure 4.40: Cell-per-thread method speed up over the grid-per-thread and row-per-thread methods for the Game of Life and clay deformation CA.

putational complexity of its rule application function, there seems to be a certain threshold on the grid size which dictates when a parallel rule application function will start to outperform the sequential rule application function. This threshold is also dependent on the number of cells processed simultaneously, the number of cells that can be processed per second by the computational device (the GPU or the CPU), and any form of overhead.

Stellenbosch University https://scholar.sun.ac.za

A sequential implementation only processes one cell at a time. However, the CPU has a higher clock speed than the GPU, and thus more cells can be processed per second for a single thread on a CPU. There is also no additional overhead required for copying data and assigning sections of the data to processing cores, as is the case with all the parallel implementations. A parallel implementation processes more than one cell simultaneously, depending on the amount of processing cores used. However, cells are processed with a lower clock speed. Thus, if the number of cells processed simultaneously do not compensate for any overhead and less throughput per thread because of a slower clock speed, the CPU will always outperform the parallel implementation, as seems to be the case with the grid-per-thread implementations. For a parallel implementation to produce a speed up over the sequential implementation, the number of cells processed simultaneously must compensate for any overhead and the lower clock speed of the GPU.

The sequential calculation time T_s for a grid size N^2 is defined as $T_s = \frac{N^2}{C_s}$, where C_s is the number of cells processed per second on the CPU. The parallel calculation time T_p for a grid size N^2 is defined as $T_p = \frac{N^2}{C_p} + O_p$, where C_p is the number of cells processed per second on the GPU and O_p is the additional overhead of the parallel implementation. At the threshold, where $T_s = T_p$, we get $\frac{N^2}{C_s} = \frac{N^2}{C_p} + O_p$. Writing this formula as a function of the grid size N^2 we get:

$$N^2 = O_p \frac{C_p C_s}{(C_p - C_s)} \tag{5.1}$$

For the parallel implementation, the number of cells processed per second is equal to the number of cells c_p processed per second by k cores; thus $C_p = kc_p$. Substituting the alternative representation for C_p into Equation 5.1 we get:

$$N^2 = O_p \frac{kc_p C_s}{(kc_p - C_s)} \tag{5.2}$$

From this equation a formula for N is determined, where the parallel implementation starts to outperform the sequential implementation:

$$N = O_p \sqrt{\frac{kc_p C_s}{(kc_p - C_s)}} \tag{5.3}$$

Equation 5.3 is represented in terms of the number of GPU cores k, the number of cells processed per GPU core per second, and the number of cells processed by the CPU per second. While the number cells processed per second stays constant, the number of GPU cores does change depending on the grid size and the segmentation method used. For the cell-per-thread segmentation method, all cores are used when the number of cells, cell blocks, or ants is equal to the number of cores.

The overhead O_p has a large influence on the value of N, depending on the amount of data that has to be copied and assigned to processing cores on the

Stellenbosch University https://scholar.sun.ac.za

GPU. Additionally, the difference in calculation time between the CPU and GPU, for certain operations such as floating point operations, also influences the overall calculation time. For example, the cell-per-thread implementation for the Game of Life CA on Computer B, starts to outperform the sequential implementation for $N = 15$ or for 225 cells. However, the cell-per-thread implementation for the clay deformation CA only starts to outperform the sequential implementation for $N = 64$ or for 4096 cells. Also note that for the Game of Life implementation, only 225 of the 2304 processing cores are used, in contrast to the 1024 cores used by the clay deformation CA. In order for the GPU to produce a speed up over the sequential implementation for the clay deformation CA, 4.55 times more GPU processing cores must be used than in the case of the Game of Life implementation.

In this chapter, the performance of a traditional sequential rule application function, for three different CA, have been measured against different parallel rule application functions. The following chapter concludes this thesis.

Stellenbosch University https://scholar.sun.ac.za

Chapter 5

Conclusion

1 Overview

For this thesis a CA framework was developed that is based on CUDA. The framework was applied to investigate the potential performance increase gained when using a GPU for parallel rule application algorithms based on different data segmentation methods, for different CA. The performance of the parallel implementations was measured against the performance of traditional sequential rule application functions of the CA implemented with the framework.

This chapter concludes the thesis by discussing the overall findings of the experiments performed and discussed in Chapter 4, and by discussing potential future work.

2 Findings

From the experiments performed on the Game of Life, clay deformation and ant clustering CA, there is a clear indication that the GPU does provide a speed up over the traditional sequential implementations of these CA. The speed up that is produced does however depend on the amount of GPU processing resources used, the complexity of the rule application function of the CA, and the grid size of the CA.

When utilizing more GPU processing resources, the speed up that is gained by the parallel implementation increases dramatically. The cell-per-thread data segmentation method, which uses all of the GPU processing resources (unless the number of cells are less than the number of processing cores), provides the best performance of the segmentation methods. This is the case for both the Game of Life and clay deformation CA, which is an expected outcome since the GPU can process more cells per instance when using all of its processing resources. When restricting the utilization of GPU processing resources, the GPU will most likely

Stellenbosch University https://scholar.sun.ac.za

not provide a speed up over the sequential implementation, as is the case with the grid-per-thread segmentation method. However, as more GPU resources are used, as is done with the row-per-thread segmentation methods, the GPU does produce a speed, but requires a larger grid size than is required for the cell-per-thread segmentation method.

The rule application functions of the clay deformation and ant clustering CA are more complex than the Game of Life rule application function. Thus, efficient parallel implementations such as the cell-per-thread or ant-per-thread segmentation methods only produce a speed up over the sequential implementation when more cells must be processed. The overhead of transferring data to and from the GPU, and assigning cells to processing cores also influence the performance of the parallel implementations. When the grid size is large, the influence of transferring data will become less significant since the sequential implementation only processes one cell at a time, whereas the parallel implementations process multiple cells simultaneously.

Finally, a parallel implementation is not necessarily the best solution for any CA, as is the case with the ant clustering CA. For a fully parallel ant clustering rule application function, compensation must be made for potential collisions. This process slows the performance of the fully parallel implementation down significantly (while using an ant-per-thread segmentation method), and the fully parallel implementation does not provide any speed up over a sequential or hybrid parallel implementations.

Ultimately, it was found that when using a GPU as a parallel processing platform for CA, a programmer must utilize as much of the GPU processing resources as possible, to produce the best possible speed up over a traditional sequential implementation of the CA. However, the programmer must be watchful of the grid size of the CA, as the overhead of data transfer and assignment, as well as the complexity of the rule application function of the particular CA does influence performance of the parallel implementation.

3 Future work

For future work the framework implemented for this thesis can be extended to test higher dimensional CA. As an example, an experiment can be conducted to test whether a three-dimensional clay deformation CA delivers the same general performance results for sequential and parallel algorithms.

Next, the performance difference between the parallel algorithms implemented on the GPU for this thesis, can be measured against different parallel algorithms performed on the CPU. Although the CPU does not have the same number of cores and cannot spawn the same number of threads as the more recent GPUs, the fact

Stellenbosch University https://scholar.sun.ac.za

that the CPU has a higher clock speed than a GPU could have a significant impact on performance difference.

Finally, the CA framework can also be used as a foundation to investigate specific real-time simulations such as general object deformation. Since this thesis confirms the advantage of using a parallel processing platform for CA rule application functions, we can investigate how efficiently the GPU simulates the deformation of, for example a glass object in real time.

Stellenbosch University https://scholar.sun.ac.za

Appendices

Stellenbosch University https://scholar.sun.ac.za

Appendix A

History of GPU architectures

Current GPU architectures allow commercial users to harness the computational ability of the GPU for all kinds of high-performance computing (HPC) projects. The primary use of the GPU is beginning to diversify. Not only does it compute realistic high resolution graphics, but it is also used in industry where data-parallelism problems occur. In this addendum an overview of past graphics architectures is given, followed by an analysis of the graphics pipeline (or rendering process) and how concepts surrounding it were rethought, in order to allow general computation on graphics hardware. Finally, a brief look into the GPU market-leading architectures, in-use today, is given.

1 Early history

During the mid 1970s, GPUs were thought of as display devices. Vendors started conceiving hardware designs around this time and some of the first display devices were:

- the CDP 1861 (or Pixie) video chip, developed by RCA Corporation;
- the Television Interface Adapter (TIA) 1A, developed by Jay Miner and integrated into the Atari 2600 gaming console;
- the MC6845 developed by Motorola, Inc.; and
- the ANTIC LSI Colour/Graphics TIA developed by Atari, Inc. and integrated into the Atari 400 gaming console [46].

These devices are only able to display low resolution images, some colour and some monochromatic. Throughout the 1980s, display devices started expanding into separate computational devices, linked to the host processor of a computer through an expansion port.

Stellenbosch University https://scholar.sun.ac.za

ATI Technologies Inc. (ATI) was founded in 1985, and started revolutionizing graphics processing, on both the commercial and technological front. The Wonder series was introduced in 1986/87 by ATI, providing higher resolution graphics at a higher colour density. With each new release the processor clock speed and device memory of the video card increased [62]. NVIDIA Corporation was founded in 1993, and introduced the NV1 chipset in 1995, which was incorporated into the Diamond Edge 3D video card. The NV1 chipset was the first commercially sold video processor that could render 3D graphics [9]. Later that same year, ATI announced the ATI Rage chipset, which was their first graphics chipset that could render 3D graphics. In 1996, ATI released the 3D RAGE I video card, utilizing the ATI Rage chipset [46].

2 From rendering to computation

In order to render any graphics onto a display device, a programmer has to be able to interface with the graphics processor. With the advent of graphics rendering application programmer interfaces (APIs), this process was made simpler. These APIs include libraries that help programmers interface with the different parts of the hardware rendering process, by relaying instructions and data to the graphics processor. Two of the most notable modern graphics rendering APIs are Silicon Graphics Inc.'s OpenGL (now managed by Khronos Group) and Microsoft's Direct3D (a sub API of Microsoft DirectX).

In the early days of 3D rendering, around 1996, the process that was followed to produce visuals, known as the *graphics pipeline*, was still in its infancy. Vertices, used to describe objects in a scene to be rendered, were first processed by the CPU, before the processed information was passed onto the graphics processor via the Peripheral Component Interconnect (PCI) bus. The graphics processor would then map colours and textures to the primitives (points, lines, triangles), generated by the CPU. Before displaying frames on the display device, the graphics processor first rasterizes the image or scene, which is the step where the image is mapped to pixels [56].

The next generation of the graphics pipeline shifted all calculation to the graphics processor, thus the CPU would only pass on scene data to the graphics processor. Other than that, the general procedure that was followed, did not change. The Accelerated Graphics Port (AGP) bus was also introduced around this time, which increased the data throughput from the CPU to the graphics processor. It was also around this time that NVIDIA introduced the term "GPU" for Graphics Processing Unit.

The first two generations of graphics pipelines were also known as fixed function pipelines. The graphics processors had dedicated engines to perform the separate tasks in the pipeline, and these engines could only be used with predefined

Stellenbosch University https://scholar.sun.ac.za

functions, as decided by the vendors, which were made accessible by the graphics processing APIs [47]. A programmer was not able to introduce custom functions and algorithms into the fixed function pipeline.

In 2001, with the introduction of NVIDIA's NV20 architecture and ATI's R200 architecture (among other vendors), the third generation graphics pipeline was established. Limited programmability was made available by introducing vertex shaders into the vector transform section of the pipeline. These shaders were programmable to a small extent, allowing the execution of basic program instructions, such as basic arithmetic. With this advancement in graphical processing technology, general purpose computation was made available [49]. However, it was difficult to do general computations, as data had to be represented as vertices, that the vertex shaders would be able to understand.

Over the next five years, up until 2006, general advances were made to GPU architectures. The most notable advancement was the introduction of pixel shaders; similar to vertex shaders but introduced into the rasterize section of the graphics pipeline. Pixel shaders, also known as programmable fragment processors, made the graphics pipeline completely programmable. True conditionals, loops, and other flow-control mechanisms were slowly integrated into the pixel and vertex shaders, as well as the ability to calculate primitives (longs and floats) with higher precision. Programmers were slowly gaining more freedom to perform more general calculations on the GPU, as well as increasing the visual quality of the graphics generated by the GPU. However, the then current GPU architectures did still use different hardware engines to perform the work in the separate section of the graphics pipeline. Because of this architectural design, calculations were not performed optimally, since there were generally more vertex shaders incorporated into the GPU to perform vertex transform operation, and less pixel shaders to perform shading and rasterizing operation. With these limitations in mind, and with GPGPU gaining more interest in the world of HPC, GPU vendors set out to re-think the entire GPU architecture [57].

3 Unified shader architecture

In 2006, the main GPU vendors NVIDIA and ATI both, released a new GPU architecture based on the unified shader architecture design. This architectural design is completely different from the previous generations. The unique processor engines of the previous generation of GPUs, designed to handle vertex, geometry, and pixel shading operations have now essentially been combined. The new design is a grid of unified data-parallel floating-point processors, also known as unified shaders or stream processors. A general enough instruction set is used to run all the different shader workloads [35]. Since 2006, the unified shader architecture has been the core design for all new GPU architectures.

Stellenbosch University https://scholar.sun.ac.za

3.1 First unified shader GPUs

ATI introduced the first GPU using unified shader architecture, known as Xenos, to the gaming console industry. This architecture was later introduced to the desktop, with the arrival of the ATI R600 architecture. NVIDIA was the first to introduced this GPU architectural design to the desktop PC, which is known as the G8x architecture. ATI pursued graphical processing with their entrance into unified shader architecture era. Although this is also the case with NVIDIA, NVIDIA also put a lot of focus into general computation. Thus, along with the NVIDIA G8x architecture, Compute Unified Device Architecture or CUDA was also introduced.

3.2 NVIDIA: Kepler

In 2012, NVIDIA launched the Kepler architecture as the successor to the Fermi architecture. For the Kepler architecture the focus is shifted to performance efficiency, and with the next generation streaming multiprocessors (SMX), NVIDIA is able to deliver three times the performance per watt. The SMXs of the Kepler architecture have 192 CUDA cores, which help to deliver increased performance when compared to the SMs of the Fermi architecture, which only have 32 CUDA cores per SM [36].

The GPUs of Computer A and Computer B, used to perform the experiments for this thesis (refer to Chapter 4), are both based on the Kepler architecture.

3.3 AMD: GCN

After AMD had finalized the acquisition of ATI in late 2006, AMD continued to produce the famous Radeon GPUs of ATI. In 2012, AMD released the Graphics Core Next (GCN) architecture as the successor to the TeraScale architecture. With the GCN architecture, AMD also set out to increase GPU performance while reducing power consumption. AMD's GCN architecture is also their first design aimed at general computing [2]. The GCN architecture, coupled with C++ AMP and OpenCL, allow programmers to use AMD GPUs based on the GCN architecture for general purpose computational tasks.

Stellenbosch University https://scholar.sun.ac.za

Appendix B

Code listings

1 Game of Life

1.1 Sequential

Listing B.1: Extract from GoL.cpp – `GoL::gridNextStateSEQ()`

```cpp
int neighbours = 0;
int state = 0;
auto t1 = high_resolution_clock::now();

for (int y = 0; y < dimy; y++) {
  for (int x = 0; x < dimx; x++) {
    neighbours = Rules::N_Moore(x,y,&gridm[0],dimx,dimy);
    state = gridm[y*dimx+x].getState();

    if ((state == 1) && (neighbours < 2 || neighbours > 3))
      //live cell is dead in next generation
      gridm_temp[y*dimx+x].updateState(0);
    else if ((state == 0) && (neighbours == 3))
      //dead cell becomes alive in next generation
      gridm_temp[y*dimx+x].updateState(1);
    else
      //cell retains its current state in next generation
      gridm_temp[y*dimx+x].updateState(state);
  }
}
grid->copyGrid(&gridm[0], &gridm_temp[0]);

auto t2 = high_resolution_clock::now();

duration<double, ratio<1, 1>> dbDuration;
dbDuration = duration_cast<duration<double>>(t2 - t1);

//return time taken to calculate next state of CA
return dbDuration.count()*1000;
```

Stellenbosch University https://scholar.sun.ac.za

Listing B.2: Extract from StaticFunctions.cpp – `Rules::N_Moore()`

```
//Counter for number of life around a cell
int life = 0;

//indexes -- out of bound row and column indexes
//Row:
int xneg = Rules::mod_n(x-1,dimx), xpos = (x+1)%dimx;
//Column:
int yneg = Rules::mod_n(y-1,dimy), ypos = (y+1)%dimy;

//Row above Cell in question
if (grid[yneg*dimx+xneg].getState() != 0) life++;
if (grid[yneg*dimx+x    ].getState() != 0) life++;
if (grid[yneg*dimx+xpos].getState() != 0) life++;

//Row in-line with Cell in question
if (grid[y*dimx+xneg].getState() != 0) life++;
if (grid[y*dimx+xpos].getState() != 0) life++;

//Row below Cell in question
if (grid[ypos*dimx+xneg].getState() != 0) life++;
if (grid[ypos*dimx+x    ].getState() != 0) life++;
if (grid[ypos*dimx+xpos].getState() != 0) life++;

return life;
```

1.2 Parallel

Listing B.3: Extract from StaticFunctions.cu – `__global__ GoL_Stencil_GpT()`

```
//Divide number of rows in the CA between number of SMs
int range = (size/smc);
if (size%smc != 0) range ++;
//Determine SM's row start and end indexes
int row_start = blockIdx.x * range;
int row_end = row_start + range;
if (row_end > size) row_end = size;

//Column: left and right column indexes:
int xneg = 0, xpos = 0, x;
//Row: upper and lower row indexes:
int yneg = 0, ypos = 0, y;

//Counter for number of life around a cell
int life = 0;
int state = 0;

for (y = row_start; y < row_end; y++) {
  //Reset next cell's neighborhood row variables
  yneg = (y+size-1)%size, ypos = (y+1)%size;
  for (x = 0; x < size; x++) {
    //Reset next cell's neighborhood column variables
    xneg = (x+size-1)%size, xpos = (x+1)%size;
    //Get cell's live neighbour count and its state
    life = N_Moore_gpu(d_in,x,xneg,xpos,y,yneg,ypos,size);
    state = d_in[y*size+x].state;
    //Apply rules to current cell
    GoL_rules_gpu(d_out,x,y,life,state,size)
  }
}
```

Stellenbosch University https://scholar.sun.ac.za

Listing B.4: Extract from StaticFunctions.cu – `__global__ GoL_Stencil_RpT()`

```
//Mapping from a thread in its block (in the grid),
//to a row in the Cellular Automaton
int y = blockIdx.x*blockDim.x + threadIdx.x;

//Check if the mapping is out of the grid-bounds
if (y >= size) return;

//Counter for number of life around a cell
int life = 0;
int state = 0;

//Column: left and right column indexes:
int xneg = 0, xpos = 0;
//Row: upper and lower row indexes:
int yneg = (y+size-1)%size, ypos = (y+1)%size;

for (int x = 0; x < size; x++) {
  //Reset next cell's neighborhood column variables
  xneg = (x+size-1)%size, xpos = (x+1)%size;
  //Get cell's live neighbour count and its state
  life = N_Moore_gpu(d_in,x,xneg,xpos,y,yneg,ypos,size);
  state = d_in[y*size+x].state;
  //Apply rules to current cell
  GoL_rules_gpu(d_out,x,y,life,state,size)
}
```

Listing B.5: Extract from StaticFunctions.cu – `__global__ GoL_Stencil_CpT()`

```
//Mapping from a thread in its block (in the grid),
//to a cell in the Cellular Automaton
int x = blockIdx.x*blockDim.x + threadIdx.x;
int y = blockIdx.y*blockDim.y + threadIdx.y;

//Check if the mapping is out of the grid-bounds
if (x >= size) return;
if (y >= size) return;

//Counter for number of life around a cell
int life = 0;
int state = 0;

//Column: left and right column indexes:
int xneg = (x+size-1)%size, xpos = (x+1)%size;
//Row: upper and lower row indexes:
int yneg = (y+size-1)%size, ypos = (y+1)%size;

//Get cell's live neighbour count and its state
life = N_Moore_gpu(d_in,x,xneg,xpos,y,yneg,ypos,size);
state = d_in[y*size+x].state;
//Apply rules to current cell
GoL_rules_gpu(d_out,x,y,life,state,size)
```

Listing B.6: Extract from StaticFunctions.cu – `__device__ GoL_Rules_GPU()`

```
//Update output array with new cell states
if (state == 1 && (life < 2 || life > 3))
  d_out[y*dimx+x].state = 0;
else if (state == 0 && life == 3)
  d_out[y*dimx+x].state = 1;
else d_out[y*dimx+x].state = d_in[y*dimx+x].state;
```

Stellenbosch University https://scholar.sun.ac.za

2 Clay deformation

2.1 Sequential

Listing B.7: Extract from clay.cpp – `Clay::gridNextStateSEQ()`

```cpp
//Perform plat push operation
pushPlate();

//Get handle on two cellular grids
auto t1 = std::chrono::high_resolution_clock::now();

//Test for stability
bool stable = false;
//Transition step counter
int trans = 0;
int r0, r1, c0, c1;

while (!stable) {
  stable = true;
  for (int y = 0; y < dimy; y+=2) {
    for (int x = 0; x < dimx; x+=2) {
      //Determine Margolus neighborhood for cell
      //If transition step is even:
      if (trans%2==0) {
        r0 = y; c0 = x;
        //Specify y-dimension barrier
        if ((y+1) < dimy) r1 = y+1;
        else r1 = r0;
        //Specify x-dimension barrier
        if ((x+1) < dimx) c1 = x+1;
        else c1 = c0;
      }
      //If transition step is odd
      else {
        r1 = y; c1 = x;
        //Specify y-dimension barrier
        if ((y-1) < 0) r0 = r1;
        else r0 = y-1;
        //Specify x-dimension barrier
        if ((x-1) < 0) c0 = c1;
        else c0 = x-1;
      }
      if (Rules::rules_clay(c0,c1,r0,r1,&gridm[0],mass_thresh,alpha,dimx)==false)
        stable = false;
    }
  }
  //grid->printGrid();
  trans++;
  if (trans > 100) break;
}
auto t2 = std::chrono::high_resolution_clock::now();

std::chrono::duration<double, std::ratio<1, 1>> dbDuration;
dbDuration = std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);

return dbDuration.count()* 1000;
```

Stellenbosch University https://scholar.sun.ac.za

Listing B.8: Extract from Rules.cpp – `Rules::rules_clay()`

```cpp
//Test if we are only working with one cell
if (x0==x1 && y0==y1) {
  if (grid[y0*dimx+x0].getMass() > thresh) return false;
  else return true;
}
int tot = (1+x1-x0)*(1+y1-y0);
int unSt = 0; int isSt = 0;

float deltaM = 0.0;
float cellM = 0.0;
float subM = 0.0;

for (int r = y0; r <= y1; r++)
  for (int c = x0; c <= x1; c++) {
    cellM = grid[r*dimx+c].getMass();
    //Test for over-burdened cells
    if (cellM > thresh) {
      subM = cellM*alpha;
      deltaM += subM;
      unSt++;
    } else if (grid[r*dimx+c].state != 2) isSt++;
    else if (grid[r*dimx+c].state == 2) tot--;
  }
//If all cells are unstable return false;
if      (unSt == tot) return false;
//Else if all cells are stable return true;
else if (isSt == tot) return true;

//ELSE some cells are stable and some not -- distribute mass

//Boolean to represent state of block being processed
bool block_stable = true;
float addM = deltaM / isSt;

for (int r = y0; r <= y1; r++)
  for (int c = x0; c <= x1; c++) {
    cellM = grid[r*dimx+c].getMass();
    if (cellM > thresh) {
      subM = cellM*alpha;
      cellM -= subM;
      grid[r*dimx+c].updateMass(cellM);
      if (cellM > thresh) block_stable = false;
    }
    else {
      cellM += addM;
      if (grid[r*dimx+c].state != 2) {
        grid[r*dimx+c].updateMass(cellM);
        grid[r*dimx+c].set(c,r,1);
      }
      if (cellM > thresh) block_stable = false;
    }
  }

return block_stable;
```

Stellenbosch University https://scholar.sun.ac.za

2.2 Parallel

Listing B.9: Extract from StaticFunctions.cu – `__global__ Clay_Stencil_GpT()`

```
//Divide the number of rows in the CA between the number of SMs
int range = dimy/smc;
if (dimy%smc != 0) range ++;
//Determine SMs row start index
int row_start = blockIdx.x * range;
int row_end = row_start + range;
if (row_end > dimy) row_end = dimy;

int r0, r1, c0, c1;
bool stable = true;
d_out[blockIdx.x] = stable;
for (int y = row_start; y < row_end; y+=2) {
  for (int x = 0; x < dimx; x+=2) {
    //Determine Margolus neighborhood for cell.
    //If transition step is even:
    if (even) {
      r0 = y; c0 = x;
      //Specify y-dimension barrier
      if ((y+1) < dimy) r1 = y+1;
      else r1 = r0;
      //Specify x-dimension barrier
      if ((x+1) < dimx) c1 = x+1;
      else c1 = c0;
    }
    //If transition step is odd
    else {
      r1 = y; c1 = x;
      //Specify y-dimension barrier
      if ((y-1) < 0) r0 = r1;
      else r0 = y-1;
      //Specify x-dimension barrier
      if ((x-1) < 0) c0 = c1;
      else c0 = x-1;
    }
    stable = Clay_Stecil_opp(d_in, dimx, r0, r1, c0, c1, thresh, alpha);
    if (stable == false)
      d_out[blockIdx.x] = false;
  }
}
```

Listing B.10: Extract from StaticFunctions.cu – `__global__ Clay_Stencil_RpT()`

```
//X/Y-indexes for d_out index
int tx = blockIdx.x*blockDim.x + threadIdx.x;
int ty = blockIdx.y*blockDim.y + threadIdx.y;
//Final index to which to write bool value to in d_out
int outIdx = ty * blockDim.x + tx;
//Y-dim stays fixed for thread:
int y = blockIdx.x*(blockDim.x*2) + (threadIdx.x*2);

//Check if the mapping is out of the grid-bounds
if (y >= dimx) return;

int r0, r1, c0, c1;

//If transition step is even:
if (even) {
  r0 = y;
```

Stellenbosch University https://scholar.sun.ac.za

```
  //Specify y-dimension barrier
  if ((y+1) < dimx) r1 = y+1;
  else r1 = r0;
} else { //If transition step is odd:
  r1 = y;
  //Specify y-dimension barrier
  if ((y-1) < 0) r0 = r1;
  else r0 = y-1;
}
bool stable = true;
d_out[outIdx] = stable;
//Start thread's work on its assigned row
for (int x = 0; x < dimx; x = x+2) {
  //If transition step is even:
  if (even) {
    c0 = x;
    //Specify x-dimension barrier
    if ((x+1) < dimx) c1 = x+1;
    else c1 = c0;
  } else { //If transition step is odd:
    c1 = x;
    //Specify x-dimension barrier
    if ((x-1) < 0) c0 = c1;
    else c0 = x-1;
  }
  stable = Clay_Stecil_opp(d_in, dimx, r0, r1, c0, c1, thresh, alpha);
  if (stable == false)
    d_out[outIdx] = false;
}
```

Listing B.11: Extract from StaticFunctions.cu – `__global__ Clay_Stencil_CpT()`

```
//Mapping from a thread in its block (in the grid),
//to a cell in the Cellular Automaton
int x  = blockIdx.x*(blockDim.x*2) + (threadIdx.x*2);
int tx = blockIdx.x*blockDim.x + threadIdx.x;
int y  = blockIdx.y*(blockDim.y*2) + (threadIdx.y*2);
int ty = blockIdx.y*blockDim.y + threadIdx.y;

//d_out index to write boolean value to
int outIdx = ty * blockDim.x + tx;

//Check if the mapping is out of the grid-bounds
if (x >= dimx) return;
if (y >= dimx) return;

int r0, r1, c0, c1;
bool stable = true;

//If transition step is even:
if (even) {
  r0 = y; c0 = x;
  //Specify y-dimension barrier
  if ((y+1) < dimx) r1 = y+1;
  else r1 = r0;
  //Specify x-dimension barrier
  if ((x+1) < dimx) c1 = x+1;
  else c1 = c0;
}
//If transition step is odd:
else {
  r1 = y; c1 = x;
  //Specify y-dimension barrier
```

Stellenbosch University https://scholar.sun.ac.za

```
  if ((y-1) < 0) r0 = r1;
  else r0 = y-1;
  //Specify x-dimension barrier
  if ((x-1) < 0) c0 = c1;
    else c0 = x-1;
  }
stable = Clay_Stecil_opp(d_in, dimx, r0, r1, c0, c1, thresh, alpha);
d_out[outIdx] = stable;
```

3 Ant clustering

3.1 Sequential

Listing B.12: Extract from antLFA.cpp – `antLFA::gridNextStateSEQ()`

```
int dx = grid->dimx; int dy = grid->dimy;

int temp_state = 0;
float r = 0.0;
float act_prb = 0.0;

Tuple retAgent;

//Get handle on two cellular grids
auto t1 = chrono::high_resolution_clock::now();

int x, y, a;
for (a = 0; a < antNum; a++) {
  x = agents[a].x; y = agents[a].y;
  temp_state = grid->getCellState(x, y);
  //TEST FOR PICKUP
  if (temp_state > 0 && agents[a].state == 0) {
    //Random real number between (0..1);
    r = (float) rand() / (float) (RAND_MAX);
    //Calculate ant's active probability
    act_prb = Rules::rules_ant_fitness(x, y, grid->grid, dx, dy, alpha);
    act_prb = Rules::rules_ant_activate(act_prb, beta, lambda);

    if (r <= act_prb) {
      //Pick up object & clear grid cell:
      agents[a].state = grid->grid[y*dx+x].state;
      grid->grid[y*dx+x].state = 0;
      agents[a].mass = grid->grid[y*dx+x].mass;
      grid->grid[y*dx+x].mass = 0.0;
    }
  }
  //TEST FOR DROP
  else if (temp_state == 0 && agents[a].state > 0) {
    //Random real number between (0..1);
    r = (float) rand() / (float) (RAND_MAX);
    //Calculate ant's active probability
    act_prb = Rules::rules_ant_fitness(x, y, grid->grid, dx, dy, alpha);
    act_prb = Rules::rules_ant_drop(act_prb, betaD);
    if (r <= act_prb) {
      //Drop object & fill grid cell:
      grid->grid[y*dx+x].state = agents[a].state;
      agents[a].state = 0;
      grid->grid[y*dx+x].mass = agents[a].mass;
```

Stellenbosch University https://scholar.sun.ac.za

```
      agents[a].mass =  0.0;
    }
  }
  //MOVE ANT TO RANDOM LOCATION
  retAgent = Rules::Ant_move_LFA(x, y, grid->grid, dx);
  if (retAgent.x != -1) {
    grid->grid[y*dx+x].sleep = false;
    agents[a].x = retAgent.x;
    agents[a].y = retAgent.y;
    grid->grid[retAgent.y*dx+retAgent.x].sleep = true;
  }
}

auto t2 = chrono::high_resolution_clock::now();

chrono::duration<double, ratio<1, 1>> dbDuration;
dbDuration = chrono::duration_cast<chrono::duration<double>>(t2 - t1);

return dbDuration.count()*1000;
```

Listing B.13: Extract from rules.cpp – `Rules::rules_ant_fitness()`

```
float fitness = 0.0;

//Variables to compute neighborhood fitness
float sum = 0.0;
float dst = 0.0;

float ai = grid[y*dimx+x].mass;
float aj = 0;

for (int j = y-1; j <= y+1; j++) {
  for (int i = x-1; i <= x+1; i++) {
    //Get state of current neighbor
    aj = grid[mod_n(j,dimy)*dimx + mod_n(i,dimx)].mass;
    //Check if neighbor is occupied and perform calculation
    if (aj != 0.0) dst = 1.0 - ((float) abs(ai - aj) / alpha);
    else dst = 0.0;

    sum += dst;
  }
}
//Final sum of neighborhood of occupied cell
sum = sum*(1.0/9.0);
//Fitness is maximum between ZERO and sum
fitness = max((float)0.0, sum);

return fitness;
```

Stellenbosch University https://scholar.sun.ac.za

Listing B.14: Extract from rules.cpp – `Rules::rules_ant_pickup()`

```
if (fitness == 0.0) return 1.0;
else return pow(beta / (beta + fitness), 2.0);
```

Listing B.15: Extract from rules.cpp – `Rules::rules_ant_drop()`

```
if (fitness < beta) return (2*fitness);
else return 1.0;
```

Listing B.16: Extract from ant.cpp – `Ant::gridNextStateSEQ()`

```
int dx = grid->dimx; int dy = grid->dimy;

int temp_state = 0;
float r = 0.0;
float act_prb = 0.0;

Tuple retAgent;

//Get handle on two cellular grids
auto t1 = chrono::high_resolution_clock::now();

int x, y, a;
for (a = 0; a < antNum; a++) {
  x = agents[a].x; y = agents[a].y;
  temp_state = grid->getCellState(x, y);
  if (temp_state > 0) {
    //Random real number between (0..1);
    r = (float) rand() / (float) (RAND_MAX);
    //Calculate ant's active probability
    act_prb = Rules::rules_ant_fitness(x, y, grid->grid, dx, dy, alpha);
    act_prb = Rules::rules_ant_activate(act_prb, beta, lambda);

    if (r <= act_prb) {
      retAgent = Rules::Ant_move_greedy(x, y, grid->grid, dx, alpha);
      if (retAgent.x != -1) {
        agents[a].x = retAgent.x;
        agents[a].y = retAgent.y;
      }
    }
  }
}

auto t2 = chrono::high_resolution_clock::now();

chrono::duration<double, ratio<1, 1>> dbDuration;
dbDuration = chrono::duration_cast<chrono::duration<double>>(t2 - t1);

return dbDuration.count()*1000;
```

Listing B.17: Extract from rules.cpp – `Rules::rules_ant_activate()`

```
if (fitness == 0.0) return 1.0;
else return pow(beta, lambda) / (pow(beta, lambda) + pow(fitness, lambda));
```

Stellenbosch University https://scholar.sun.ac.za

Listing B.18: Extract from rules.cpp – `Rules::Ant_move_greedy()`

```
bool movable = false;
//Fitness verify
float fitness = -1;
float ftemp = 0.0;

int state  = grid[y*dim+x].state;
float mass = grid[y*dim+x].mass;

Tuple agent; agent.x = -1; agent.y = -1;
int mi, mj;

int z = 1;
//Iterate of all neighbours:
for (int j = -1*z; j <= z; j=j+1) {
  for (int i = -1*z; i <= z; i=i+1) {
    if (j != 0) {
      //Check if neighboring cell is empty
      if (grid[mod_n(y+j,dim) * dim + mod_n(x+i,dim)].state == 0) {
        ftemp = rules_ant_fitness(mod_n(x+i,dim), mod_n(y+j,dim), grid, dim,
            dim, alpha);
        if (ftemp > fitness) {
          fitness = ftemp;
          mi = x; mj = mod_n(y+j,dim);
          agent.x = mi; agent.y = mj;
          movable = true;
        }
      }
    }
  }
}
//Check if there are no positions to move to:
if (!movable) {
  agent.x = -1; agent.y = -1;
  return agent;
} else { //Move ant to best location
  grid[mj * dim + mi].state = state;
  grid[mj * dim + mi].mass  = mass;
  grid[y*dim+x].state = 0; grid[y*dim+x].mass = 0.0;
}
return agent;
```

3.2 Parallel

Listing B.19: Extract from antLFA.cpp – `antLFA::gridNextStatePARhybrid()`

```
int dx = grid->dimx;

int temp_state = 0;
float r = 0.0;
float act_prb = 0.0;
Tuple retAgent;

//Get handle on two cellular grids
auto t1 = chrono::high_resolution_clock::now();

//Call GPU prep function which initiates activate probability calculation
Rules::AntFitnessCUDA(grid->getGridStartingIndex(), d_in, &agents[0], d_out,
    byteCA, byteANT, dx, antNum, alpha, beta, betaD, lambda, true);
int x, y, a;
```

Stellenbosch University https://scholar.sun.ac.za

```
for (a = 0; a < antNum; a++) {
  x = agents[a].x; y = agents[a].y;
  temp_state = grid->getCellState(x, y);
  //Get activate probability
  act_prb = agents[a].f;
  //TEST FOR PICKUP
  if (temp_state > 0 && agents[a].state == 0) {
    //Random real number between (0..1);
    r = (float) rand() / (float) (RAND_MAX);
    if (r <= act_prb) {
      //Pick up object & clear grid cell:
      agents[a].state = grid->grid[y*dx+x].state;
      grid->grid[y*dx+x].state = 0;
      agents[a].mass = grid->grid[y*dx+x].mass;
      grid->grid[y*dx+x].mass = 0.0;
    }
  }
  //TEST FOR DROP
  else if (temp_state == 0 && agents[a].state > 0) {
    //Random real number between (0..1);
    r = (float) rand() / (float) (RAND_MAX);
    if (r <= act_prb) {
      //Drop object & fill grid cell:
      grid->grid[y*dx+x].state = agents[a].state;
      agents[a].state = 0;
      grid->grid[y*dx+x].mass = agents[a].mass;
      agents[a].mass =  0.0;
    }
  }
  if (agents[a].state > 0) retAgent = Rules::Ant_move_greedy_LFA(x, y, grid->grid,
      dx, alpha);
  else retAgent = Rules::Ant_move_LFA(x, y, grid->grid, dx);
  if (retAgent.x != -1) {
    grid->grid[y*dx+x].sleep = false;
    agents[a].x = retAgent.x;
    agents[a].y = retAgent.y;
    grid->grid[retAgent.y*dx+retAgent.x].sleep = true;
  }
}
auto t2 = chrono::high_resolution_clock::now();
chrono::duration<double, ratio<1, 1>> dbDuration;
dbDuration = chrono::duration_cast<chrono::duration<double>>(t2 - t1);

return dbDuration.count()*1000;
```

Listing B.20: Extract from StaticFunctions.cu – `__global__ Ant_Fitness_ApT()`

```
//Mapping from a thread to an ant
int a = blockIdx.x*blockDim.x + threadIdx.x;

//Check if the mapping is out of the lattice-bounds
if (a >= ants) return;

int x = d_ants[a].x; int y = d_ants[a].y;

float fit = rules_ant_fitness(x, y, d_grid, dim, A);
d_ants[a].f = fit;
```

Listing B.21: Extract from StaticFunctions.cu – `__global__ Ant_LFAprob_ApT()`

```
//Mapping from a thread to an ant
int a = blockIdx.x*blockDim.x + threadIdx.x;
```

Stellenbosch University https://scholar.sun.ac.za

```

//Check if the mapping is out of the lattice-bounds
if (a >= ants) return;

int x = d_ants[a].x; int y = d_ants[a].y;
float fit;

if (d_grid[y*dim+x].state > 0 && d_ants[a].state == 0)
  fit = rules_ant_pckPrb(d_ants[a].f, beta);
else if (d_grid[y*dim+x].state == 0 && d_ants[a].state > 0)
  fit = rules_ant_drpPrb(d_ants[a].f, betaD);
d_ants[a].f = fit;
```

Listing B.22: Extract from ant.cpp – `ant::gridNextStatePARhybrid()`

```
int dx = grid->dimx;

float r = 0.0;
float act_prb = 0.0;
Tuple retAgent;

//Get handle on two cellular grids
auto t1 = chrono::high_resolution_clock::now();

//Call GPU prep function which initiates activate probability calculation
Rules::AntFitnessCUDA(grid->getGridStartingIndex(), d_in, &agents[0], d_out,
    byteCA, byteANT, dx, antNum, alpha, beta, 0.0, lambda, false);

int x, y, a;
for (a = 0; a < antNum; a++) {
  x = agents[a].x; y = agents[a].y;
  //grid->getCellState(x, y);
  //Get activate probability
  act_prb = agents[a].f;

  //Random real number between (0..1);
  r = (float) rand() / (float) (RAND_MAX);

  if (r <= act_prb) {
    retAgent = Rules::Ant_move_greedy(x, y, grid->grid, dx, alpha);
    if (retAgent.x != -1) {
      agents[a].x = retAgent.x;
      agents[a].y = retAgent.y;
    }
  }
}
auto t2 = chrono::high_resolution_clock::now();
chrono::duration<double, ratio<1, 1>> dbDuration;
dbDuration = chrono::duration_cast<chrono::duration<double>>(t2 - t1);

return dbDuration.count()*1000;
```

Listing B.23: Extract from StaticFunctions.cu – `__global__ Ant_ActPrb_ApT()`

```
//Mapping from a thread to an ant
int a = blockIdx.x*blockDim.x + threadIdx.x;
//Check if the mapping is out of the lattice-bounds
if (a >= ants) return;

float fit = d_ants[a].f;
fit = rules_ant_actPrb(fit, B, L);
d_ants[a].f = fit;
```

Stellenbosch University https://scholar.sun.ac.za

Bibliography

[1] Altera (2013). Stratix 10 FPGAs and SoCs: delivering the unimaginable.
Available at: http://www.altera.com/devices/fpga/stratix-fpgas/stratix10/stx10-index.jsp 4

[2] AMD (2012). GCN Architecture.
Available at: http://www.amd.com/en-us/innovations/software-technologies/gcn 98

[3] Arata, H., Takai, Y., Takai, N. and Yamamoto, T. (1999). Free-form shape modeling by 3D cellular automata. In: *Proceedings of the Shape Modeling and Applications International Conference*, pp. 242–247. IEEE.
Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=749346 14, 30, 45

[4] Baldwin, C.Y. and Clark, K.B. (2000). *The Power of Modularity*, vol. 1 of *Design Rules*. The MIT Press. 14

[5] Buck, I. (2008). Parallel programming with CUDA. In: *Proceedings of the SC08 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–42.
Available at: http://gpgpu.org/static/sc2008/M02-02_CUDA.pdf 12, 23, 28

[6] Butts, M. (2007). Synchronization through communication in a massively parallel processor array. *IEEE Micro*, vol. 27, no. 05, pp. 32–40.
Available at: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4378781 6

[7] Caux, J., Hill, D. and Siregar, P. (2011). *Accelerating 3D cellular automata computation with GP-GPU in the context of integrative biology.*
Available at: http://hal.archives-ouvertes.fr/hal-00679045 8

[8] Chen, L., Xu, X. and Chen, Y. (2007 January). A novel ant clustering algorithm based on cellular automata. *Web Intelligence and Agent Systems*, vol. 5, no. 1, pp. 1–14.
Available at: http://www.cse.wustl.edu/~ychen/public/IAT.pdf 63, 65

Stellenbosch University https://scholar.sun.ac.za

[9] Dandumont, P. (2008 July). 13 years of Nvidia graphics cards.
Available at: http://www.tomshardware.com/picturestory/464-nvidia-graphics-cards.html 96

[10] Druon, S., Crosnier, A. and Brigandat, L. (2003). Efficient cellular automata for 2D/3D free-form modeling. In: *Proceedings of the 11-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, vol. 11, pp. 102–108. WCSG.
Available at: https://otik.zcu.cz/handle/11025/1675 45

[11] Fan, Z., Qiu, F., Kaufman, A. and Yoakum-Stover, S. (2004). GPU cluster for high performance computing. In: *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*.
Available at: http://dl.acm.org/citation.cfm?id=1049991 9

[12] Ferrando, N., Gosálvez, M., Cerdá, J., Gadea, R. and Sato, K. (2011 March). Octree-based, GPU implementation of a continuous cellular automaton for the simulation of complex, evolving surfaces. *Computer Physics Communications*, vol. 182, no. 3, pp. 628–640.
Available at: http://linkinghub.elsevier.com/retrieve/pii/S0010465510004509 8

[13] Ganguly, N., Sikdar, B. and Deutsch, A. (2003). A survey on cellular automata. pp. 1–30.
Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.107.7729 3

[14] Gardner, M. (1970 oct). Mathematical games: The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, vol. 223, no. 4, pp. 120–123.
Available at: http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm 30, 32

[15] Gibson, M.J., Keedwell, E.C. and Savić, D.a. (2014 November). An investigation of the efficient implementation of cellular automata on multi-core CPU and GPU hardware. *Journal of Parallel and Distributed Computing*, pp. 1–15.
Available at: http://linkinghub.elsevier.com/retrieve/pii/S0743731514002044 8

[16] Gobron, S., Bonafos, H. and Mestre, D. (2008). GPU accelerated computation and visualization of hexagonal cellular automata. *Cellular Automata*, vol. 5191, pp. 512–521.
Available at: http://link.springer.com/chapter/10.1007/978-3-540-79992-4_67 8

[17] Gobron, S., Devillard, F. and Heit, B. (2007 dec). Retina simulation using cellular automata and GPU programming. *Machine Vision and Applications*, vol. 18, no. 6,

Stellenbosch University https://scholar.sun.ac.za

pp. 331–342.
Available at: http://link.springer.com/article/10.1007/s00138-006-0065-8 8

[18] Gosálvez, M., Xing, Y., Sato, K. and Nieminen, R. (2009 October). Discrete and continuous cellular automata for the simulation of propagating surfaces. *Sensors and Actuators A: Physical*, vol. 155, no. 1, pp. 98–112.
Available at: http://linkinghub.elsevier.com/retrieve/pii/S0924424709003756 3, 8

[19] Halmstad College (2009). The Ambric processor array – a first look.
Available at: http://www.hh.se/download/18.70cf2e49129168da0158000120470/1341267676645/The+Ambric+Processor+Array.pdf ix, 6

[20] Harris, M. and Coombe, G. (2002). Physically-based visual simulation on graphics hardware. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 109–118.
Available at: http://dl.acm.org/citation.cfm?id=569061 9

[21] Hindriksen, V. (2012). OpenCL vs CUDA misconceptions.
Available at: http://streamcomputing.eu/blog/2011-06-22/opencl-vs-cuda-misconceptions 12

[22] Ilachinski, A. (2001). *Cellular Automata: A Discrete Universe.* World Scientific Publishing Co. 3

[23] Kauffmann, C. and Piché, N. (2008). Cellular automaton for ultra-fast watershed transform on GPU. *19th International Conference on Pattern Recognition*, pp. 1–4.
Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4761628 7

[24] Kauffmann, C. and Piché, N. (2010). Seeded ND medical image segmentation by cellular automaton on GPU. *International Journal of Computer Assisted Radiology and Surgery*, vol. 5, no. 3, pp. 251–262.
Available at: http://www.ncbi.nlm.nih.gov/pubmed/20033502 1, 7

[25] Khronos (2014). The open standard for parallel programming of heterogeneous systems.
Available at: https://www.khronos.org/opencl 9, 10

[26] Kirtzic, J., Allen, D. and Daescu, O. (2013). Applying the parallel GPU model to radiation therapy treatment. In: *Proceedings of the 2013 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 1–7.
Available at: http://world-comp.org/p2013/PDP3098.pdf 9

[27] Klöckner, A. (2013). CUDA vs OpenCL: Which should I use?
Available at: http://wiki.tiker.net/CudaVsOpenCL 12

Stellenbosch University https://scholar.sun.ac.za

[28] Larsen, E.S. and McAllister, D. (2001). Fast matrix multiplies using graphics hardware. In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pp. 43–48. ACM Press, New York City, New York, USA.
Available at: http://portal.acm.org/citation.cfm?doid=582034.582089 9

[29] Lopez-Torres, M., Guisado, J., Jiménez-Morales, F. and Diaz-del Rio, F. (2012). GPU-based cellular automata simulations of laser dynamics.
Available at: http://www.jornadassarteco.org/js2012/papers/paper_151.pdf 8

[30] Luebke, D. and Owens, J. (2013). Intro to parallel programming.
Available at: https://www.udacity.com/course/cs344 22, 23, 24

[31] Lumer, E.D. and Faieta, B. (1994). Diversity and adaptation in populations of clustering ants. In: *Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, SAB94, pp. 501–508. MIT Press. 30, 63, 66

[32] Millo, J. and Simone, R.D. (2013). Explicit routing schemes for implementation of cellular automata on processor arrays. *Natural Computing*, vol. 12, no. 3, pp. 353–368.
Available at: http://link.springer.com/article/10.1007/s11047-013-9378-5 6

[33] Murtaza, S., Hoekstra, A. and Sloot, P. (2009). Compute bound and I/O bound cellular automata simulations on FPGA logic. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 1, no. 4, pp. 1–21.
Available at: http://dl.acm.org/citation.cfm?id=1462592 ix, 4, 5

[34] Nandi, S. and Kar, B. (1994). Theory and applications of cellular automata in cryptography. *IEEE Transactions on Computers*, vol. 43, no. 12, pp. 1346–1357.
Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=338094 3

[35] Navarro, C.A., Hitschfeld-Kahler, N. and Mateu, L. (2013). A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329.
Available at: http://www.global-sci.com/readabs.php?vol=15&page=285&issue=2&ppage=329&year=2014 7, 97

[36] NVIDIA (2012). Next-generation CUDA compute architecture.
Available at: http://www.nvidia.com/object/nvidia-kepler.html 98

[37] NVIDIA (2014 August). *CUDA C Programming Guide.*
Available at: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf 22, 23, 24

[38] NVIDIA (2014). CUDA zone.
Available at: https://developer.nvidia.com/cuda-zone 12

Stellenbosch University https://scholar.sun.ac.za

[39] NVIDIA (2014). What is CUDA?
Available at: http://www.nvidia.com/object/cuda_home_new.html 9, 11

[40] Owens, J. and Luebke, D. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113.
Available at: http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2007.01012.x/full 9

[41] Peterson, G.L., Mayer, C.B. and Kubler, T.L. (2008 June). Ant clustering with locally weighted ant perception and diversified memory. *Swarm Intelligence*, vol. 2, no. 1, pp. 43–68.
Available at: http://link.springer.com/10.1007/s11721-008-0011-7 63, 64

[42] Rybacki, S., Himmelspach, J. and Uhrmacher, A.M. (2009). Experiments with single core, multi-core, and GPU based computation of cellular automata. *First International Conference on Advances in System Simulation*, pp. 62–67.
Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5283985 8

[43] Schiff, J.L. (2011). *Cellular Automata: A Discrete View of the World*. John Wiley & Sons, Inc. 3

[44] Shi, L., Liu, W. and Zhang, H. (2012). A survey of GPU-based medical image computing techniques. *Quantitative Imaging in Medicine and Surgery*, vol. 2, no. 3, pp. 188–206.
Available at: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3496509&tool=pmcentrez&rendertype=abstract;http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3496509 9

[45] Shiffman, D. (2012). *The Nature of Code*. 1st edn. Creative Commons. 1, 3, 14, 32

[46] Singer, G. (2013 March). The history of the modern graphics processor.
Available at: http://www.techspot.com/article/650-history-of-the-gpu 95, 96

[47] Teschner, M. (2013). Image processing and computer graphics: Rendering pipeline. Tech. Rep., Albert-Ludwigs-University of Freiburg.
Available at: http://cg.informatik.uni-freiburg.de/course_notes/graphics_01_pipeline.pdf 97

[48] Thomas, D.B., Howes, L. and Luk, W. (2009). A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. *Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays – FPGA '09*, pp. 63–72.
Available at: http://portal.acm.org/citation.cfm?doid=1508128.1508139 4, 6

Stellenbosch University https://scholar.sun.ac.za

[49] Thompson, C. and Oskin, M. (2002). Using modern graphics architectures for general-purpose computing: a framework and analysis. In: *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 306–317.
Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1176259 9, 97

[50] Toffoli, T. and Margolus, N. (1987). *Cellular Automata Machines: A New Environment for Modeling*. MIT Press. 45

[51] Top, P. and Gokhale, M. (2009). Application experiments: MPPA and FPGA. In: *Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pp. 291–294.
Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5290896 7

[52] Tosaka (2008). CUDA processing flow (En).
Available at: http://en.wikipedia.org/wiki/File:CUDA_processing_flow_(En).PNG ix, 11

[53] Tran, J. and Jordan, D. (2004). New challenges for cellular automata simulation on the GPU.
Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.131.9597&rep=rep1&type=pdf 3, 4

[54] Trendall, C. and Stewart, A.J. (2000). General calculations using graphics hardware, with application to interactive caustics. In: *Proceedings of the Eurographics Workshop on Rendering Techniques*, pp. 287–298. 9

[55] Trevett, N. (2013). OpenCL Introduction. In: *Proceedings of the SIGGRAPH Asia 2013 Conference*, pp. 1–21. ACM.
Available at: http://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf ix, 10

[56] Venkatasubramanian, S. (2014). Understanding the graphics pipeline. Tech. Rep., University of Pennsylvania.
Available at: http://www.seas.upenn.edu/~cis565/LECTURES/Lecture2%20New.pdf 96

[57] Verschelde, J. (2012 March). Evolution of graphics pipelines.
Available at: http://homepages.math.uic.edu/~jan/mcs572s12 97

[58] Wolfram, S. (1982). Cellular automata as simple self-organizing systems. pp. 1–12.
Available at: http://cds.cern.ch/record/140047 1

[59] Wolfram, S. (1983). Cellular automata. *Los Alamos Science*, pp. 2–21.
Available at: http://dl.acm.org/citation.cfm?id=1098682 1

Stellenbosch University https://scholar.sun.ac.za

[60] Wolfram, S. (1985). Twenty problems in the theory of cellular automata. *Physica Scripta*, vol. T9, pp. 170–183.
Available at: http://iopscience.iop.org/1402-4896/1985/T9/029 3

[61] Wolfram, S. (1988). Cellular automaton supercomputing. *High-speed computing: scientific applications and algorithm design*, pp. 499–509.
Available at: http://www.stephenwolfram.com/pdf/Cellular-Automaton-Supercomputing-Stephen-Wolfram-Article.pdf 1

[62] Woodster, J. (2006 July). Full ATI timeline 1984–2006.
Available at: http://pycckuu.blogspot.com/2006/07/full-ati-timeline-1984-2006.html 96

[63] wxWidgets (2014). Cross-Platform GUI Library.
Available at: https://www.wxwidgets.org 19

[64] Yang, R. and Welch, G. (2002). Fast image segmentation and smoothing using commodity graphics hardware. *Journal of Graphics Tools*, vol. 7, no. 4, pp. 91–100.
Available at: http://www.tandfonline.com/doi/abs/10.1080/10867651.2002.10487576 9

[65] Yuen, A. and Kay, R. (2009). Applications of cellular automata.
Available at: http://www.cs.bham.ac.uk/~rjh/courses/NatureInspiredDesign/2009-10/StudentWork/Group2/design-report.pdf 3

[66] Zaloudek, L., Sekanina, L. and Simek, V. (2009). GPU accelerators for evolvable cellular automata. In: *Proceedings of the 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pp. 533–537. IEEE Computer Society.
Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5359646 8